Arm® A64 Instruction Set

for A-profile architecture

DDI 0596 (ID070122)

Arm A64 Instruction Set
for A-profile architecture

Release Information

For information on the change history and known issues for this release, see the **Release Notes** in the **A64 ISA XML for A-profile architecture (2022-06)**.

Proprietary Notice

Arm Limited. Company 02557590 registered in England.
110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349 version 21.0)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Product Status

The information relating to all Armv9-A features (including FEAT_RME and FEAT_SME) and the Armv8-A features, except for the Optional 64-bit external interface to the Performance Monitors, is at **Beta quality**. Beta quality means that all major features of the specification are described, some details might be missing.

The information relating to the Optional 64-bit external interface to the Performance Monitors is at **Alpha quality**. Alpha quality means that most major features of the specification are described in the manual, some features and details might be missing.

Web Address

http://www.arm.com

Progressive Terminology Commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used terms that can be offensive. Arm strives to lead the industry and create change.

Previous issues of this document included terms that can be offensive. We have replaced these terms. If you find offensive terms in this document, please contact terms@arm.com.

A64 -- Base Instructions (alphabetic order)

ADC: Add with Carry.

ADCS: Add with Carry, setting flags.

ADD (extended register): Add (extended register).

ADD (immediate): Add (immediate).

ADD (shifted register): Add (shifted register).

ADDG: Add with Tag.

ADDS (extended register): Add (extended register), setting flags.

ADDS (immediate): Add (immediate), setting flags.

ADDS (shifted register): Add (shifted register), setting flags.

ADR: Form PC-relative address.

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

AND (immediate): Bitwise AND (immediate).

AND (shifted register): Bitwise AND (shifted register).

ANDS (immediate): Bitwise AND (immediate), setting flags.

ANDS (shifted register): Bitwise AND (shifted register), setting flags.

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

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

ASRV: Arithmetic Shift Right Variable.

AT: Address Translate: an alias of SYS.

AUTDA, AUTDZA: Authenticate Data address, using key A.

AUTDB, AUTDZB: Authenticate Data address, using key B.

AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA: Authenticate Instruction address, using key A.

AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB: Authenticate Instruction address, using key B.

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

B: Branch.

B.cond: Branch conditionally.

BC.cond: Branch Consistent conditionally.

BFC: Bitfield Clear: an alias of BFM.

BFI: Bitfield Insert: an alias of BFM.

BFM: Bitfield Move.

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

BIC (shifted register): Bitwise Bit Clear (shifted register).

BICS (shifted register): Bitwise Bit Clear (shifted register), setting flags.

BL: Branch with Link.

BLR: Branch with Link to Register.

BLRAA, BLRAAZ, BLRAB, BLRABZ: Branch with Link to Register, with pointer authentication.

BR: Branch to Register.

BRAA, BRAAZ, BRAB, BRABZ: Branch to Register, with pointer authentication.

BRB: Branch Record Buffer: an alias of SYS.

BRK: Breakpoint instruction.

BTI: Branch Target Identification.

CAS, CASA, CASAL, CASL: Compare and Swap word or doubleword in memory.

CASB, CASAB, CASALB, CASLB: Compare and Swap byte in memory.

CASH, CASAH, CASALH, CASLH: Compare and Swap halfword in memory.

CASP, CASPA, CASPAL, CASPL: Compare and Swap Pair of words or doublewords in memory.

CBNZ: Compare and Branch on Nonzero.

CBZ: Compare and Branch on Zero.

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

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

CCMP (immediate): Conditional Compare (immediate).

CCMP (register): Conditional Compare (register).

CFINV: Invert Carry Flag.

CFP: Control Flow Prediction Restriction by Context: an alias of SYS.

CINC: Conditional Increment: an alias of CSINC.

CINV: Conditional Invert: an alias of CSINV.

CLREX: Clear Exclusive.

CLS: Count Leading Sign bits.

CLZ: Count Leading Zeros.

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

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

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

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

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

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

CMPP: Compare with Tag: an alias of SUBPS.

CNEG: Conditional Negate: an alias of CSNEG.

CPP: Cache Prefetch Prediction Restriction by Context: an alias of SYS.

CPYFP, CPYFM, CPYFE: Memory Copy Forward-only.

CPYFPN, CPYFMN, CPYFEN: Memory Copy Forward-only, reads and writes non-temporal.

CPYFPRN, CPYFMRN, CPYFERN: Memory Copy Forward-only, reads non-temporal.

CPYFPRT, CPYFMRT, CPYFERT: Memory Copy Forward-only, reads unprivileged.

CPYFPRTN, CPYFMRTN, CPYFERTN: Memory Copy Forward-only, reads unprivileged, reads and writes non-temporal.

CPYFPRTRN, CPYFMRTRN, CPYFERTRN: Memory Copy Forward-only, reads unprivileged and non-temporal.

CPYFPRTWN, CPYFMRTWN, CPYFERTWN: Memory Copy Forward-only, reads unprivileged, writes non-temporal.

CPYFPT, CPYFMT, CPYFET: Memory Copy Forward-only, reads and writes unprivileged.

CPYFPTN, CPYFMTN, CPYFETN: Memory Copy Forward-only, reads and writes unprivileged and non-temporal.

CPYFPTRN, CPYFMTRN, CPYFETRN: Memory Copy Forward-only, reads and writes unprivileged, reads non-temporal.

CPYFPTWN, CPYFMTWN, CPYFETWN: Memory Copy Forward-only, reads and writes unprivileged, writes non-temporal.

CPYFPWN, CPYFMWN, CPYFEWN: Memory Copy Forward-only, writes non-temporal.

CPYFPWT, CPYFMWT, CPYFEWT: Memory Copy Forward-only, writes unprivileged.

CPYFPWTN, CPYFMWTN, CPYFEWTN: Memory Copy Forward-only, writes unprivileged, reads and writes non-temporal.

CPYFPWTRN, CPYFMWTRN, CPYFEWTRN: Memory Copy Forward-only, writes unprivileged, reads non-temporal.

CPYFPWTWN, CPYFMWTWN, CPYFEWTWN: Memory Copy Forward-only, writes unprivileged and non-temporal.

CPYP, CPYM, CPYE: Memory Copy.

CPYPN, CPYMN, CPYEN: Memory Copy, reads and writes non-temporal.

CPYPRN, CPYMRN, CPYERN: Memory Copy, reads non-temporal.

CPYPRT, CPYMRT, CPYERT: Memory Copy, reads unprivileged.

CPYPRTN, CPYMRTN, CPYERTN: Memory Copy, reads unprivileged, reads and writes non-temporal.

CPYPRTRN, CPYMRTRN, CPYERTRN: Memory Copy, reads unprivileged and non-temporal.

CPYPRTWN, CPYMRTWN, CPYERTWN: Memory Copy, reads unprivileged, writes non-temporal.

CPYPT, CPYMT, CPYET: Memory Copy, reads and writes unprivileged.

CPYPTN, CPYMTN, CPYETN: Memory Copy, reads and writes unprivileged and non-temporal.

CPYPTRN, CPYMTRN, CPYETRN: Memory Copy, reads and writes unprivileged, reads non-temporal.

CPYPTWN, CPYMTWN, CPYETWN: Memory Copy, reads and writes unprivileged, writes non-temporal.

CPYPWN, CPYMWN, CPYEWN: Memory Copy, writes non-temporal.

CPYPWT, CPYMWT, CPYEWT: Memory Copy, writes unprivileged.

CPYPWTN, CPYMWTN, CPYEWTN: Memory Copy, writes unprivileged, reads and writes non-temporal.

CPYPWTRN, CPYMWTRN, CPYEWTRN: Memory Copy, writes unprivileged, reads non-temporal.

CPYPWTWN, CPYMWTWN, CPYEWTWN: Memory Copy, writes unprivileged and non-temporal.

CRC32B, CRC32H, CRC32W, CRC32X: CRC32 checksum.

CRC32CB, CRC32CH, CRC32CW, CRC32CX: CRC32C checksum.

CSDB: Consumption of Speculative Data Barrier.

CSEL: Conditional Select.

CSET: Conditional Set: an alias of CSINC.

CSETM: Conditional Set Mask: an alias of CSINV.

CSINC: Conditional Select Increment.

CSINV: Conditional Select Invert.

CSNEG: Conditional Select Negation.

DC: Data Cache operation: an alias of SYS.

DCPS1: Debug Change PE State to EL1..

DCPS2: Debug Change PE State to EL2..

DCPS3: Debug Change PE State to EL3.

DGH: Data Gathering Hint.

DMB: Data Memory Barrier.

DRPS: Debug restore process state.

DSB: Data Synchronization Barrier.

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

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

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

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

ERET: Exception Return.

ERETAA, ERETAB: Exception Return, with pointer authentication.

ESB: Error Synchronization Barrier.

EXTR: Extract register.

GMI: Tag Mask Insert.

HINT: Hint instruction.

HLT: Halt instruction.

HVC: Hypervisor Call.

IC: Instruction Cache operation: an alias of SYS.

IRG: Insert Random Tag.

ISB: Instruction Synchronization Barrier.

LD64B: Single-copy Atomic 64-byte Load.

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

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

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

LDAPR: Load-Acquire RCpc Register.

LDAPRB: Load-Acquire RCpc Register Byte.

LDAPRH: Load-Acquire RCpc Register Halfword.

LDAPUR: Load-Acquire RCpc Register (unscaled).

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

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

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

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

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

LDAR: Load-Acquire Register.

LDARB: Load-Acquire Register Byte.

LDARH: Load-Acquire Register Halfword.

LDAXP: Load-Acquire Exclusive Pair of Registers.

LDAXR: Load-Acquire Exclusive Register.

LDAXRB: Load-Acquire Exclusive Register Byte.

LDAXRH: Load-Acquire Exclusive Register Halfword.

LDCLR, LDCLRA, LDCLRAL, LDCLRL: Atomic bit clear on word or doubleword in memory.

LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB: Atomic bit clear on byte in memory.

LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH: Atomic bit clear on halfword in memory.

LDEOR, LDEORA, LDEORAL, LDEORL: Atomic exclusive OR on word or doubleword in memory.

LDEORB, LDEORAB, LDEORALB, LDEORLB: Atomic exclusive OR on byte in memory.

LDEORH, LDEORAH, LDEORALH, LDEORLH: Atomic exclusive OR on halfword in memory.

LDG: Load Allocation Tag.

LDGM: Load Tag Multiple.

LDLAR: Load LOAcquire Register.

LDLARB: Load LOAcquire Register Byte.

LDLARH: Load LOAcquire Register Halfword.

LDNP: Load Pair of Registers, with non-temporal hint.

LDP: Load Pair of Registers.

LDPSW: Load Pair of Registers Signed Word.

LDR (immediate): Load Register (immediate).

LDR (literal): Load Register (literal).

LDR (register): Load Register (register).

LDRAA, LDRAB: Load Register, with pointer authentication.

LDRB (immediate): Load Register Byte (immediate).

LDRB (register): Load Register Byte (register).

LDRH (immediate): Load Register Halfword (immediate).

LDRH (register): Load Register Halfword (register).

LDRSB (immediate): Load Register Signed Byte (immediate).

LDRSB (register): Load Register Signed Byte (register).

LDRSH (immediate): Load Register Signed Halfword (immediate).

LDRSH (register): Load Register Signed Halfword (register).

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

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

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

LDSET, LDSETA, LDSETAL, LDSETL: Atomic bit set on word or doubleword in memory.

LDSETB, LDSETAB, LDSETALB, LDSETLB: Atomic bit set on byte in memory.

LDSETH, LDSETAH, LDSETALH, LDSETLH: Atomic bit set on halfword in memory.

LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL: Atomic signed maximum on word or doubleword in memory.

LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB: Atomic signed maximum on byte in memory.

LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH: Atomic signed maximum on halfword in memory.

LDSMIN, LDSMINA, LDSMINAL, LDSMINL: Atomic signed minimum on word or doubleword in memory.

LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB: Atomic signed minimum on byte in memory.

LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH: Atomic signed minimum on halfword in memory.

LDTR: Load Register (unprivileged).

LDTRB: Load Register Byte (unprivileged).

LDTRH: Load Register Halfword (unprivileged).

LDTRSB: Load Register Signed Byte (unprivileged).

LDTRSH: Load Register Signed Halfword (unprivileged).

LDTRSW: Load Register Signed Word (unprivileged).

LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL: Atomic unsigned maximum on word or doubleword in memory.

LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB: Atomic unsigned maximum on byte in memory.

LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH: Atomic unsigned maximum on halfword in memory.

LDUMIN, LDUMINA, LDUMINAL, LDUMINL: Atomic unsigned minimum on word or doubleword in memory.

LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB: Atomic unsigned minimum on byte in memory.

LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH: Atomic unsigned minimum on halfword in memory.

LDUR: Load Register (unscaled).

LDURB: Load Register Byte (unscaled).

LDURH: Load Register Halfword (unscaled).

LDURSB: Load Register Signed Byte (unscaled).

LDURSH: Load Register Signed Halfword (unscaled).

LDURSW: Load Register Signed Word (unscaled).

LDXP: Load Exclusive Pair of Registers.

LDXR: Load Exclusive Register.

LDXRB: Load Exclusive Register Byte.

LDXRH: Load Exclusive Register Halfword.

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

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

LSLV: Logical Shift Left Variable.

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

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

LSRV: Logical Shift Right Variable.

MADD: Multiply-Add.

MNEG: Multiply-Negate: an alias of MSUB.

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

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

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

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

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

MOVK: Move wide with keep.

MOVN: Move wide with NOT.

MOVZ: Move wide with zero.

MRS: Move System Register.

MSR (immediate): Move immediate value to Special Register.

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

MSUB: Multiply-Subtract.

MUL: Multiply: an alias of MADD.

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

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

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

NGC: Negate with Carry: an alias of SBC.

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

NOP: No Operation.

ORN (shifted register): Bitwise OR NOT (shifted register).

ORR (immediate): Bitwise OR (immediate).

ORR (shifted register): Bitwise OR (shifted register).

PACDA, PACDZA: Pointer Authentication Code for Data address, using key A.

PACDB, PACDZB: Pointer Authentication Code for Data address, using key B.

PACGA: Pointer Authentication Code, using Generic key.

PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA: Pointer Authentication Code for Instruction address, using key A.

PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB: Pointer Authentication Code for Instruction address, using key B.

PRFM (immediate): Prefetch Memory (immediate).

PRFM (literal): Prefetch Memory (literal).

PRFM (register): Prefetch Memory (register).

PRFUM: Prefetch Memory (unscaled offset).

PSB CSYNC: Profiling Synchronization Barrier.

PSSBB: Physical Speculative Store Bypass Barrier: an alias of DSB.

RBIT: Reverse Bits.

RET: Return from subroutine.

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

REV: Reverse Bytes.

REV16: Reverse bytes in 16-bit halfwords.

REV32: Reverse bytes in 32-bit words.

REV64: Reverse Bytes: an alias of REV.

RMIF: Rotate, Mask Insert Flags.

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

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

RORV: Rotate Right Variable.

SB: Speculation Barrier.

SBC: Subtract with Carry.

SBCS: Subtract with Carry, setting flags.

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

SBFM: Signed Bitfield Move.

SBFX: Signed Bitfield Extract: an alias of SBFM.

SDIV: Signed Divide.

SETF8, SETF16: Evaluation of 8 or 16 bit flag values.

SETGP, SETGM, SETGE: Memory Set with tag setting.

SETGPN, SETGMN, SETGEN: Memory Set with tag setting, non-temporal.

SETGPT, SETGMT, SETGET: Memory Set with tag setting, unprivileged.

SETGPTN, SETGMTN, SETGETN: Memory Set with tag setting, unprivileged and non-temporal.

SETP, SETM, SETE: Memory Set.

SETPN, SETMN, SETEN: Memory Set, non-temporal.

SETPT, SETMT, SETET: Memory Set, unprivileged.

SETPTN, SETMTN, SETETN: Memory Set, unprivileged and non-temporal.

SEV: Send Event.

SEVL: Send Event Local.

SMADDL: Signed Multiply-Add Long.

SMC: Secure Monitor Call.

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

SMSTART: Enables access to Streaming SVE mode and SME architectural state: an alias of MSR (immediate).

SMSTOP: Disables access to Streaming SVE mode and SME architectural state: an alias of MSR (immediate).

SMSUBL: Signed Multiply-Subtract Long.

SMULH: Signed Multiply High.

SMULL: Signed Multiply Long: an alias of SMADDL.

SSBB: Speculative Store Bypass Barrier: an alias of DSB.

ST2G: Store Allocation Tags.

ST64B: Single-copy Atomic 64-byte Store without Return.

ST64BV: Single-copy Atomic 64-byte Store with Return.

ST64BV0: Single-copy Atomic 64-byte EL0 Store with Return.

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

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

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

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

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

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

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

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

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

STG: Store Allocation Tag.

STGM: Store Tag Multiple.

STGP: Store Allocation Tag and Pair of registers.

STLLR: Store LORelease Register.

STLLRB: Store LORelease Register Byte.

STLLRH: Store LORelease Register Halfword.

STLR: Store-Release Register.

STLRB: Store-Release Register Byte.

STLRH: Store-Release Register Halfword.

STLUR: Store-Release Register (unscaled).

STLURB: Store-Release Register Byte (unscaled).

STLURH: Store-Release Register Halfword (unscaled).

STLXP: Store-Release Exclusive Pair of registers.

STLXR: Store-Release Exclusive Register.

STLXRB: Store-Release Exclusive Register Byte.

STLXRH: Store-Release Exclusive Register Halfword.

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

STP: Store Pair of Registers.

STR (immediate): Store Register (immediate).

STR (register): Store Register (register).

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

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

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

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

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

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

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

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

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

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

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

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

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

STTR: Store Register (unprivileged).

STTRB: Store Register Byte (unprivileged).

STTRH: Store Register Halfword (unprivileged).

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

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

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

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

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

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

STUR: Store Register (unscaled).

STURB: Store Register Byte (unscaled).

STURH: Store Register Halfword (unscaled).

STXP: Store Exclusive Pair of registers.

STXR: Store Exclusive Register.

STXRB: Store Exclusive Register Byte.

STXRH: Store Exclusive Register Halfword.

STZ2G: Store Allocation Tags, Zeroing.

STZG: Store Allocation Tag, Zeroing.

STZGM: Store Tag and Zero Multiple.

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

SUB (immediate): Subtract (immediate).

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

SUBG: Subtract with Tag.

SUBP: Subtract Pointer.

SUBPS: Subtract Pointer, setting Flags.

SUBS (extended register): Subtract (extended register), setting flags.

SUBS (immediate): Subtract (immediate), setting flags.

SUBS (shifted register): Subtract (shifted register), setting flags.

SVC: Supervisor Call.

SWP, SWPA, SWPAL, SWPL: Swap word or doubleword in memory.

SWPB, SWPAB, SWPALB, SWPLB: Swap byte in memory.

SWPH, SWPAH, SWPALH, SWPLH: Swap halfword in memory.

SXTB: Signed Extend Byte: an alias of SBFM.

SXTH: Sign Extend Halfword: an alias of SBFM.

SXTW: Sign Extend Word: an alias of SBFM.

SYS: System instruction.

SYSL: System instruction with result.

TBNZ: Test bit and Branch if Nonzero.

TBZ: Test bit and Branch if Zero.

TCANCEL: Cancel current transaction.

TCOMMIT: Commit current transaction.

TLBI: TLB Invalidate operation: an alias of SYS.

TSB CSYNC: Trace Synchronization Barrier.

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

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

TSTART: Start transaction.

TTEST: Test transaction state.

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

UBFM: Unsigned Bitfield Move.

UBFX: Unsigned Bitfield Extract: an alias of UBFM.

UDF: Permanently Undefined.

UDIV: Unsigned Divide.

UMADDL: Unsigned Multiply-Add Long.

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

UMSUBL: Unsigned Multiply-Subtract Long.

UMULH: Unsigned Multiply High.

UMULL: Unsigned Multiply Long: an alias of UMADDL.

UXTB: Unsigned Extend Byte: an alias of UBFM.

UXTH: Unsigned Extend Halfword: an alias of UBFM.

WFE: Wait For Event.

WFET: Wait For Event with Timeout.

WFI: Wait For Interrupt.

WFIT: Wait For Interrupt with Timeout.

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

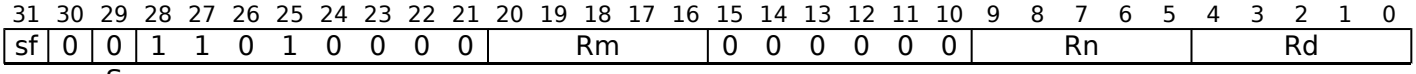
XPACD, XPACI, XPACLRI: Strip Pointer Authentication Code.

YIELD: YIELD.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ADC

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



32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

Assembler Symbols

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

Operation

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

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

X[d, datasize] = result;
```

Operational information

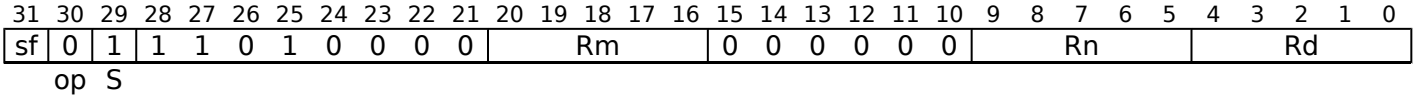
If PSTATE.DIT is 1:

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ADCS

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



32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

Assembler Symbols

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

Operation

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

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

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

X[d, datasize] = result;
```

Operational information

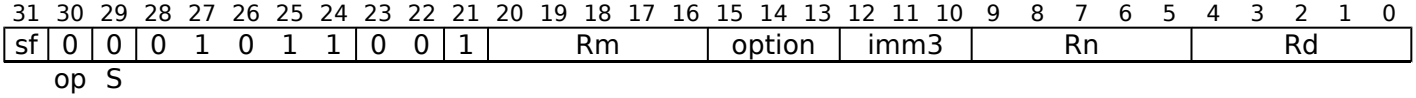
If PSTATE.DIT is 1:

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ADD (extended register)

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



32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

Assembler Symbols

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[]<datasize-1:0> else X[n, datasize];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift, datasize);

(result, -) = AddWithCarry(operand1, operand2, '0');

if d == 31 then
    SP[] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

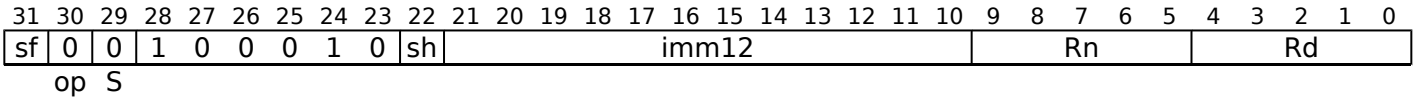
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ADD (immediate)

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

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



32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

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

Assembler Symbols

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

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

Alias Conditions

Alias	Is preferred when
MOV (to/from SP)	`sh == '0' && imm12 == '000000000000' && (Rd == '11111' || Rn == '11111')`

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[]<datasize-1:0> else X[n, datasize];

(result, -) = AddWithCarry(operand1, imm, '0');

if d == 31 then
    SP[] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

Operational information

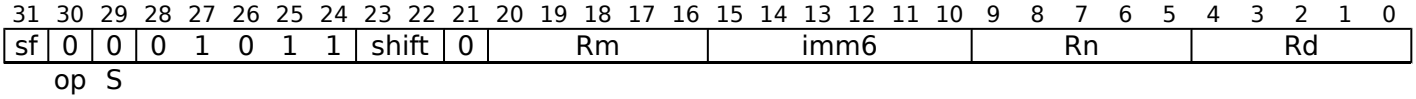
If PSTATE.DIT is 1:

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ADD (shifted register)

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



32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

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

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

Assembler Symbols

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

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

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

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

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

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

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

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

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

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

Operation

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

(result, -) = AddWithCarry(operand1, operand2, '0');

X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

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

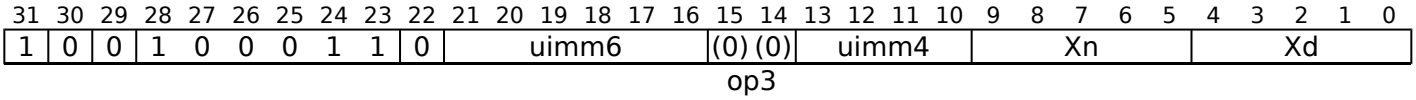
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ADDG

Add with Tag adds an immediate value scaled by the Tag granule to the address in the source register, modifies the Logical Address Tag of the address using an immediate value, and writes the result to the destination register. Tags specified in GCR_EL1.Exclude are excluded from the possible outputs when modifying the Logical Address Tag.

Integer

(FEAT_MTE)



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

```
if !HaveMTEExt() then UNDEFINED;
integer d = UInt(Xd);
integer n = UInt(Xn);
bits(64) offset = LSL(ZeroExtend(uimm6, 64), LOG2_TAG_GRANULE);
```

Assembler Symbols

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

Operation

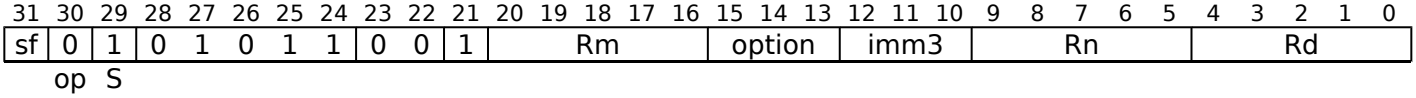
```
bits(64) operand1 = if n == 31 then SP[] else X[n, 64];
bits(4) start_tag = AArch64.AllocationTagFromAddress(operand1);
bits(16) exclude = GCR_EL1.Exclude;
bits(64) result;
bits(4) rtag;

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

(result, -) = AddWithCarry(operand1, offset, '0');
result = AArch64.AddressWithAllocationTag(result, AccType_NORMAL, rtag);

if d == 31 then
    SP[] = result;
else
    X[d, 64] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ADDS (extended register)

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

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



32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

Assembler Symbols

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Alias Conditions

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

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[]<datasize-1:0> else X[n, datasize];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift, datasize);
bits(4) nzcv;

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

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

X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

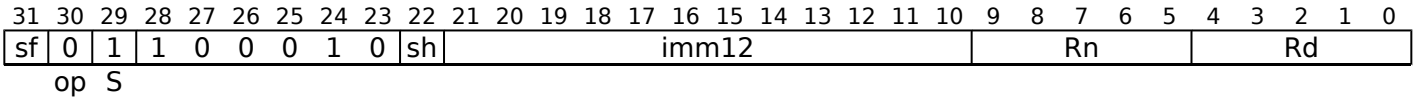
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ADDS (immediate)

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

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



32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

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

Assembler Symbols

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

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

Alias Conditions

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

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[]<datasize-1:0> else X[n, datasize];
bits(4) nzcv;

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

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

X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

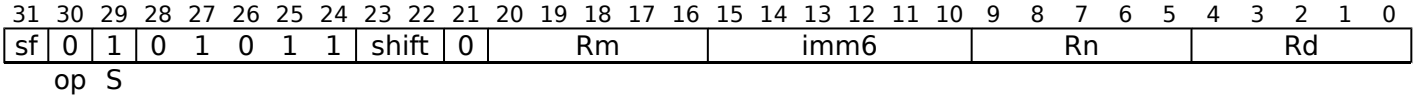
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ADDS (shifted register)

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

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



32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

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

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

Assembler Symbols

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

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

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

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

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

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

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

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

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

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

Alias Conditions

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

Operation

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

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

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

X[d, datasize] = result;
```

Operational information

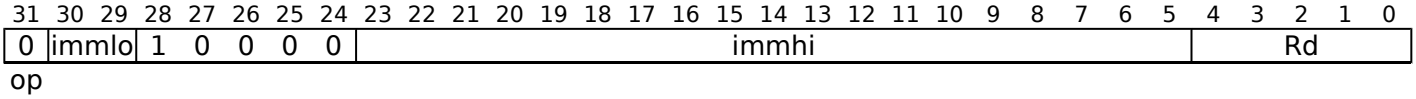
If PSTATE.DIT is 1:

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ADR

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



ADR <Xd>, <label>

```
integer d = UInt(Rd);
bits(64) imm;

imm = SignExtend(immhi:immlo, 64);
```

Assembler Symbols

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

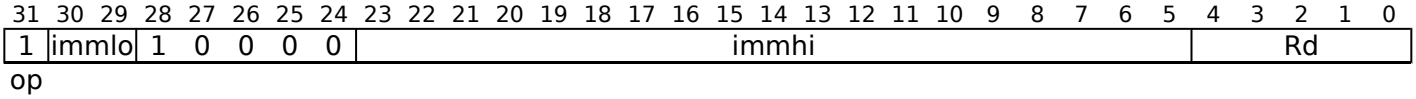
Operation

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

X[d, 64] = base + imm;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ADRP

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



ADRP <Xd>, <label>

```
integer d = UInt(Rd);
bits(64) imm;

imm = SignExtend(immhi:immlo:Zeros(12), 64);
```

Assembler Symbols

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

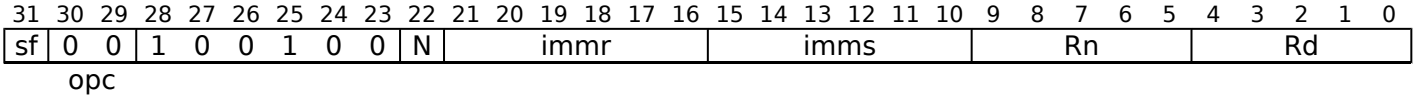
Operation

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

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

X[d, 64] = base + imm;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

AND (immediate)

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



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

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

64-bit (sf == 1)

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE, datasize);
```

Assembler Symbols

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

Operation

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

result = operand1 AND imm;
if d == 31 then
    SP[] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

Operational information

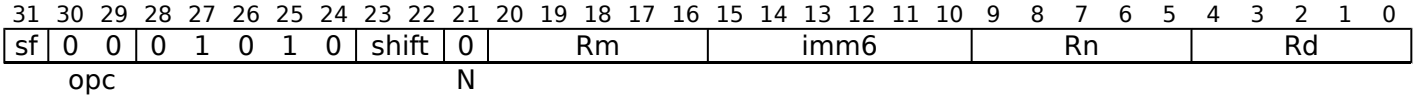
If PSTATE.DIT is 1:

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

AND (shifted register)

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



32-bit (sf == 0)

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

64-bit (sf == 1)

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

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

Assembler Symbols

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

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

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

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

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

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

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

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

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

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

Operation

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

result = operand1 AND operand2;
X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

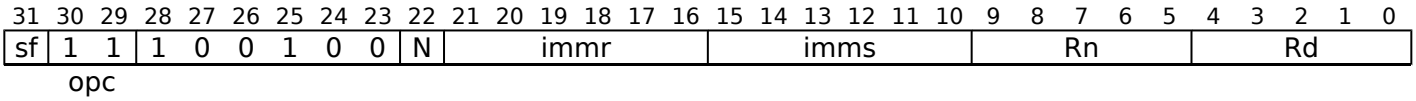
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ANDS (immediate)

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

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



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

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

64-bit (sf == 1)

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

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

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

Assembler Symbols

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

Alias Conditions

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

Operation

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

result = operand1 AND imm;
PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d, datasize] = result;
```

Operational information

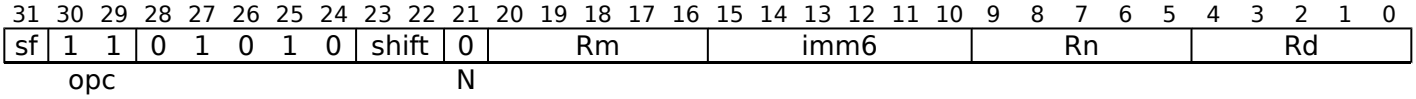
If PSTATE.DIT is 1:

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

ANDS (shifted register)

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

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



32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

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

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

Assembler Symbols

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

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

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

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

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

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

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

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

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

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

Alias Conditions

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

Operation

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

result = operand1 AND operand2;
PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

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

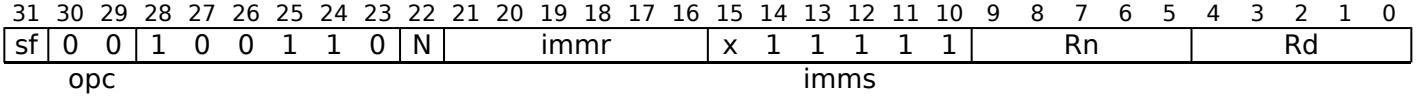
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ASR (immediate)

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

This is an alias of SBFM. This means:

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



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

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

is equivalent to

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

and is always the preferred disassembly.

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

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

is equivalent to

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

and is always the preferred disassembly.

Assembler Symbols

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

Operation

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

Operational information

If PSTATE.DIT is 1:

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

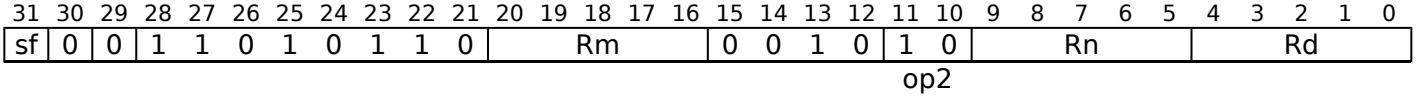
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ASR (register)

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

This is an alias of ASRV. This means:

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



32-bit (sf == 0)

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

is equivalent to

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

and is always the preferred disassembly.

64-bit (sf == 1)

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

is equivalent to

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

and is always the preferred disassembly.

Assembler Symbols

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

Operation

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

Operational information

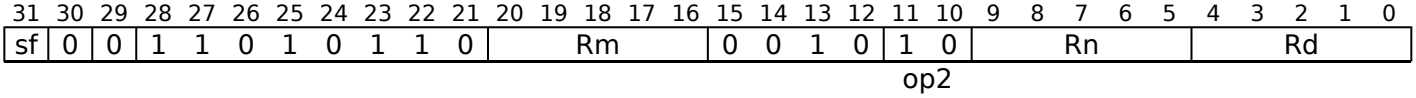
If PSTATE.DIT is 1:

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

ASRV

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

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



32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

Assembler Symbols

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

Operation

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

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

Operational information

If PSTATE.DIT is 1:

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

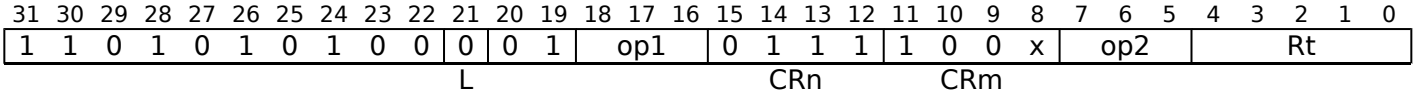
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

AT

Address Translate. For more information, see *op0==0b01, cache maintenance, TLB maintenance, and address translation instructions*.

This is an alias of SYS. This means:

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



AT <at_op>, <Xt>

is equivalent to

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

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

Assembler Symbols

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

op1	CRm<0>	op2	<at_op>	Architectural Feature
000	0	000	S1E1R	-
000	0	001	S1E1W	-
000	0	010	S1E0R	-
000	0	011	S1E0W	-
000	1	000	S1E1RP	FEAT_PAN2
000	1	001	S1E1WP	FEAT_PAN2
100	0	000	S1E2R	-
100	0	001	S1E2W	-
100	0	100	S12E1R	-
100	0	101	S12E1W	-
100	0	110	S12E0R	-
100	0	111	S12E0W	-
110	0	000	S1E3R	-
110	0	001	S1E3W	-

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

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

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

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

Operation

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

AUTDA, AUTDZA

Authenticate Data address, using key A. This instruction authenticates a data address, using a modifier and key A. The address is in the general-purpose register that is specified by <Xd>.

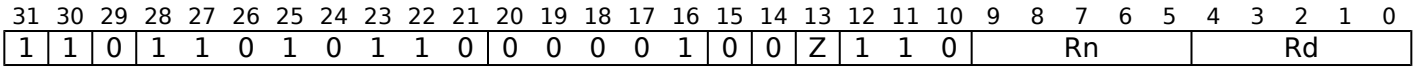
The modifier is:

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

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

Integer

(FEAT_PAuth)



AUTDA (Z == 0)

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

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

```
AUTDZA <Xd>
```

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

if !HavePACExt() then
    UNDEFINED;

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

Assembler Symbols

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

Operation

```
if HavePACExt() then
    if source_is_sp then
        X[d, 64] = AuthDA(X[d, 64], SP[], FALSE);
    else
        X[d, 64] = AuthDA(X[d, 64], X[n, 64], FALSE);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

AUTDB, AUTDZB

Authenticate Data address, using key B. This instruction authenticates a data address, using a modifier and key B. The address is in the general-purpose register that is specified by <Xd>.

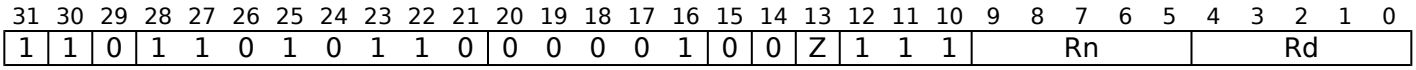
The modifier is:

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

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

Integer

(FEAT_PAuth)



AUTDB (Z == 0)

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

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

```
AUTDZB <Xd>
```

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

if !HavePACExt() then
    UNDEFINED;

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

Assembler Symbols

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

Operation

```
if HavePACExt() then
    if source_is_sp then
        X[d, 64] = AuthDB(X[d, 64], SP[], FALSE);
    else
        X[d, 64] = AuthDB(X[d, 64], X[n, 64], FALSE);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA

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

The address is:

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

The modifier is:

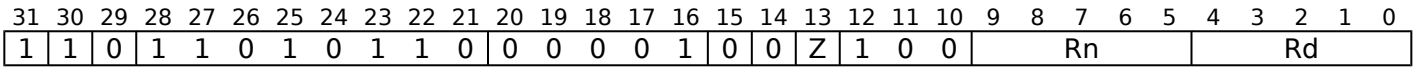
- In the general-purpose register or stack pointer that is specified by <Xn|SP> for AUTIA.
- The value zero, for AUTIZA and AUTIAZ.
- In X16, for AUTIA1716.
- In SP, for AUTIASP.

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

It has encodings from 2 classes: Integer and System

Integer

(FEAT_PAuth)



AUTIA (Z == 0)

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

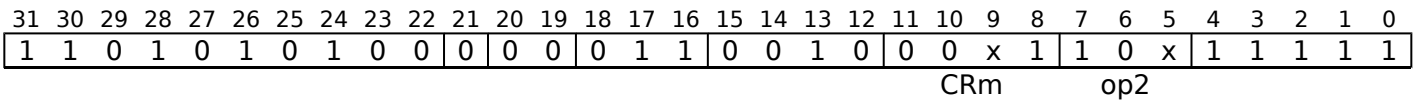
AUTIZA (Z == 1 && Rn == 11111)

```
AUTIZA <Xd>
```

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

if !HavePACExt() then
    UNDEFINED;

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

System

(FEAT_PAuth)



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

AUTIA1716

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

AUTIASP

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

AUTIAZ

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

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

Assembler Symbols

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

Operation

```
if HavePACExt() then
    if source_is_sp then
        X[d, 64] = AuthIA(X[d, 64], SP[], FALSE);
    else
        X[d, 64] = AuthIA(X[d, 64], X[n, 64], FALSE);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB

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

The address is:

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

The modifier is:

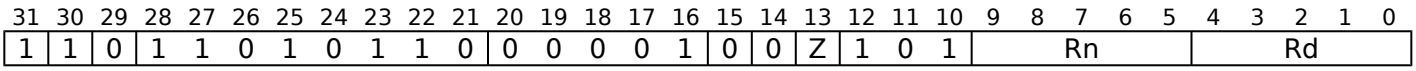
- In the general-purpose register or stack pointer that is specified by <Xn|SP> for AUTIB.
- The value zero, for AUTIZB and AUTIBZ.
- In X16, for AUTIB1716.
- In SP, for AUTIBSP.

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

It has encodings from 2 classes: Integer and System

Integer

(FEAT_PAuth)



AUTIB (Z == 0)

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

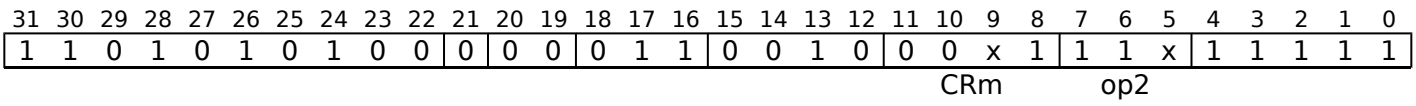
AUTIZB (Z == 1 && Rn == 11111)

```
AUTIZB <Xd>
```

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

if !HavePACExt() then
    UNDEFINED;

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

System

(FEAT_PAuth)



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

AUTIB1716

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

AUTIBSP

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

AUTIBZ

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

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

Assembler Symbols

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

Operation

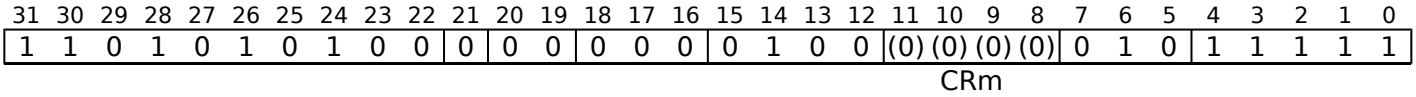
```
if HavePACExt() then
    if source_is_sp then
        X[d, 64] = AuthIB(X[d, 64], SP[], FALSE);
    else
        X[d, 64] = AuthIB(X[d, 64], X[n, 64], FALSE);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

AXFLAG

Convert floating-point condition flags from Arm to external format. This instruction converts the state of the PSTATE.{N,Z,C,V} flags from a form representing the result of an Arm floating-point scalar compare instruction to an alternative representation required by some software.

System
(FEAT_FlagM2)



AXFLAG

```
if !HaveFlagFormatExt() then UNDEFINED;
```

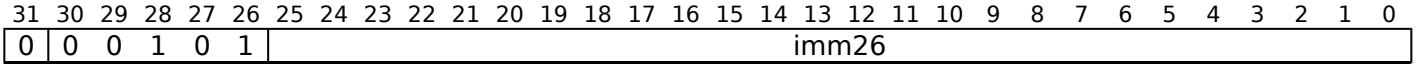
Operation

```
bit z = PSTATE.Z OR PSTATE.V;
bit c = PSTATE.C AND NOT(PSTATE.V);

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

B

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



B <label>

```
bits(64) offset = SignExtend(imm26:'00', 64);
```

Assembler Symbols

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

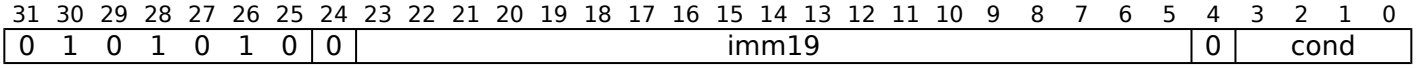
Operation

```
BranchTo(PC[] + offset, BranchType_DIR, FALSE);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

B.cond

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



B.<cond> <label>

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

Assembler Symbols

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

Operation

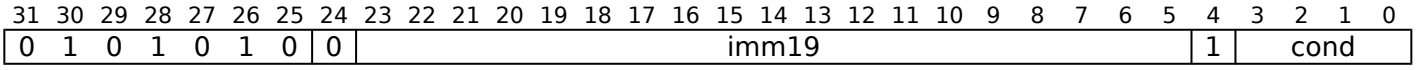
```
if ConditionHolds(cond) then
    BranchTo(PC[] + offset, BranchType_DIR, TRUE);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BC.cond

Branch Consistent conditionally to a label at a PC-relative offset, with a hint that this branch will behave very consistently and is very unlikely to change direction.

19-bit signed PC-relative branch offset
(FEAT_HBC)



BC.<cond> <label>

```
if !HaveFeatHBC() then UNDEFINED;
bits(64) offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

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

Operation

```
if ConditionHolds(cond) then
    BranchTo(PC[] + offset, BranchType_DIR, TRUE);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

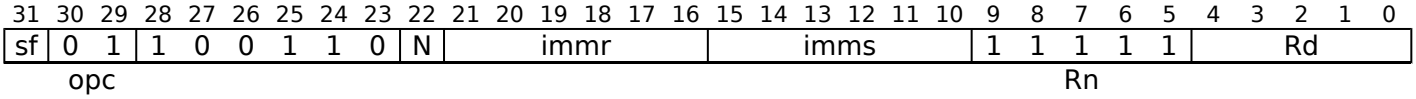
BFC

Bitfield Clear sets a bitfield of <width> bits at bit position <lsb> of the destination register to zero, leaving the other destination bits unchanged.

This is an alias of BFM. This means:

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

Leaving other bits unchanged
(FEAT_ASMv8p2)



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

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

is equivalent to

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

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

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

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

is equivalent to

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

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

Assembler Symbols

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

Operation

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

Operational information

If PSTATE.DIT is 1:

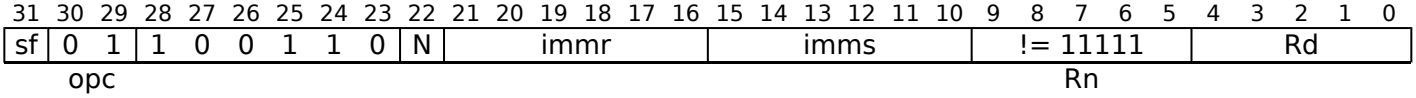
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

BFI

Bitfield Insert copies a bitfield of <width> bits from the least significant bits of the source register to bit position <lsb> of the destination register, leaving the other destination bits unchanged.

This is an alias of BFM. This means:

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



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

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

is equivalent to

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

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

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

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

is equivalent to

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

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

Assembler Symbols

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

Operation

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

Operational information

If PSTATE.DIT is 1:

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

BFM

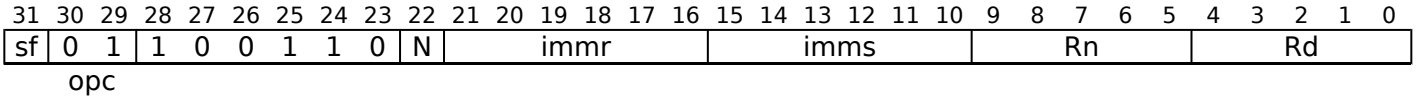
Bitfield Move is usually accessed via one of its aliases, which are always preferred for disassembly.

If <imms> is greater than or equal to <immr>, this copies a bitfield of (<imms>-<immr>+1) bits starting from bit position <immr> in the source register to the least significant bits of the destination register.

If <imms> is less than <immr>, this copies a bitfield of (<imms>+1) bits from the least significant bits of the source register to bit position (regsize-<immr>) of the destination register, where regsize is the destination register size of 32 or 64 bits.

In both cases the other bits of the destination register remain unchanged.

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



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

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

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

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

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

integer r;
bits(datasize) wmask;
bits(datasize) tmask;

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

r = UInt(immr);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE, datasize);
```

Assembler Symbols

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

Alias Conditions

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

Operation

```
bits(datasize) dst = X[d, datasize];
bits(datasize) src = X[n, datasize];

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

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

Operational information

If PSTATE.DIT is 1:

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

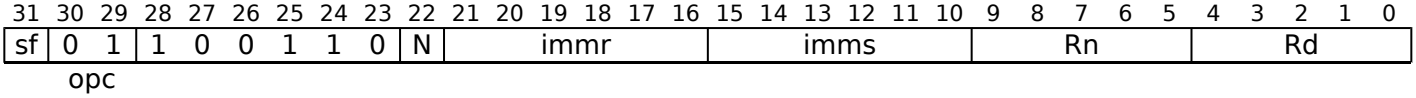
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BFXIL

Bitfield Extract and Insert Low copies a bitfield of <width> bits starting from bit position <lsb> in the source register to the least significant bits of the destination register, leaving the other destination bits unchanged.

This is an alias of BFM. This means:

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



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

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

is equivalent to

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

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

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

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

is equivalent to

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

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

Assembler Symbols

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

Operation

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

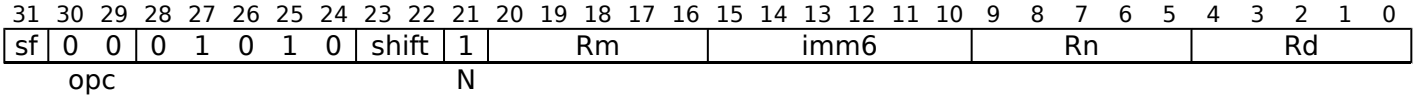
Operational information

If PSTATE.DIT is 1:

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

BIC (shifted register)

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



32-bit (sf == 0)

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

64-bit (sf == 1)

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

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

Assembler Symbols

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

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

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

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

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

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

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

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

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

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

Operation

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

operand2 = NOT(operand2);

result = operand1 AND operand2;
X[d, datasize] = result;
```

Operational information

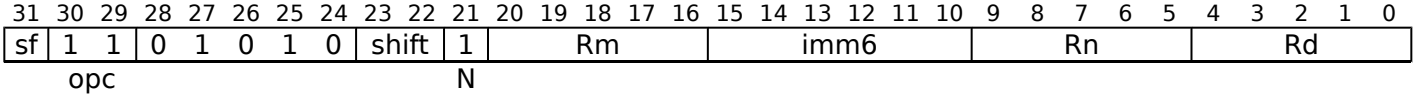
If PSTATE.DIT is 1:

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BICS (shifted register)

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



32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

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

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

Assembler Symbols

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

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

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

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

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

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

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

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

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

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

Operation

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

operand2 = NOT(operand2);

result = operand1 AND operand2;
PSTATE.<N,Z,C,V> = result<datasize-1>:IsZeroBit(result):'00';

X[d, datasize] = result;
```

Operational information

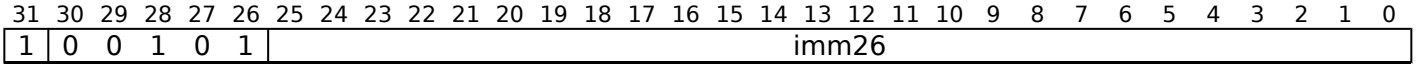
If PSTATE.DIT is 1:

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BL

Branch with Link branches to a PC-relative offset, setting the register X30 to PC+4. It provides a hint that this is a subroutine call.



BL <label>

```
bits(64) offset = SignExtend(imm26:'00', 64);
```

Assembler Symbols

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

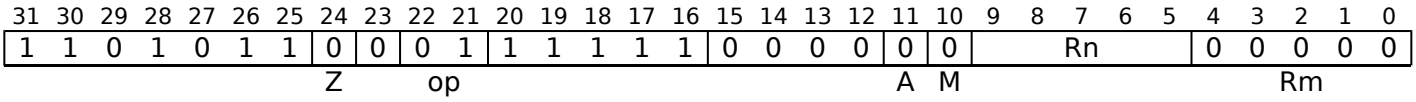
Operation

```
X[30, 64] = PC[] + 4;

BranchTo(PC[] + offset, BranchType_DIRCALL, FALSE);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BLR

Branch with Link to Register calls a subroutine at an address in a register, setting register X30 to PC+4.



BLR <Xn>

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

Assembler Symbols

<Xn>	Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

Operation

```
bits(64) target = X[n, 64];

X[30, 64] = PC[] + 4;

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '10';
BranchTo(target, BranchType_INDCALL, FALSE);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BLRAA, BLRAAZ, BLRAB, BLRABZ

Branch with Link to Register, with pointer authentication. This instruction authenticates the address in the general-purpose register that is specified by <Xn>, using a modifier and the specified key, and calls a subroutine at the authenticated address, setting register X30 to PC+4.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xm|SP> for BLRAA and BLRAB.
- The value zero, for BLRAAZ and BLRABZ.

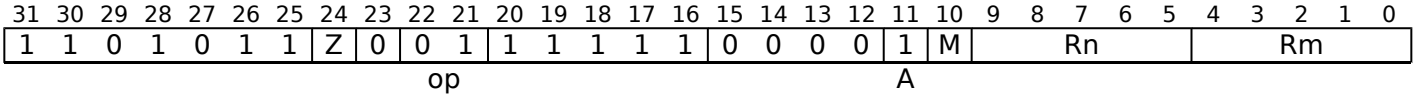
Key A is used for BLRAA and BLRAAZ, and key B is used for BLRAB and BLRABZ.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to the general-purpose register.

Integer

(FEAT_PAuth)



Key A, zero modifier (Z == 0 && M == 0 && Rm == 11111)

```
BLRAAZ <Xn>
```

Key A, register modifier (Z == 1 && M == 0)

```
BLRAA <Xn>, <Xm|SP>
```

Key B, zero modifier (Z == 0 && M == 1 && Rm == 11111)

```
BLRABZ <Xn>
```

Key B, register modifier (Z == 1 && M == 1)

```
BLRAB <Xn>, <Xm|SP>
```

```
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean use_key_a = (M == '0');
boolean source_is_sp = ((Z == '1') && (m == 31));

if !HavePACExt() then
    UNDEFINED;

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

Assembler Symbols

<Xn>	Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.
<Xm|SP>	Is the 64-bit name of the general-purpose source register or stack pointer holding the modifier, encoded in the "Rm" field.

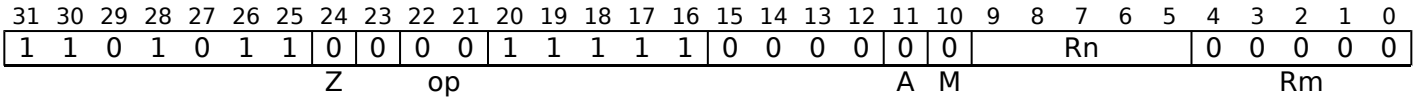
Operation

```
bits(64) target = X[n, 64];

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

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

X[30, 64] = PC[] + 4;

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '10';
BranchTo(target, BranchType_INDCALL, FALSE);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BR

Branch to Register branches unconditionally to an address in a register, with a hint that this is not a subroutine return.



BR <Xn>

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

Assembler Symbols

<Xn>	Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

Operation

```
bits(64) target = X[n, 64];

// Value in BTypeNext will be used to set PSTATE.BTYPE
if InGuardedPage then
    if n == 16 || n == 17 then
        BTypeNext = '01';
    else
        BTypeNext = '11';
else
    BTypeNext = '01';
BranchTo(target, BranchType_INDIR, FALSE);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BRAA, BRAAZ, BRAB, BRABZ

Branch to Register, with pointer authentication. This instruction authenticates the address in the general-purpose register that is specified by <Xn>, using a modifier and the specified key, and branches to the authenticated address.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xm|SP> for BRAA and BRAB.
- The value zero, for BRAAZ and BRABZ.

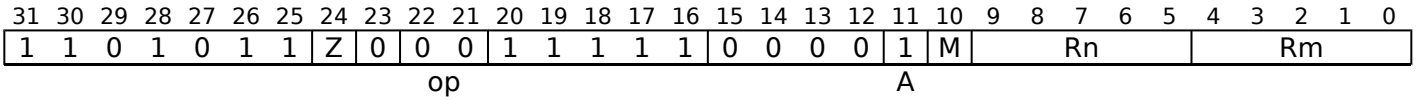
Key A is used for BRAA and BRAAZ, and key B is used for BRAB and BRABZ.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to the general-purpose register.

Integer

(FEAT_PAuth)



Key A, zero modifier (Z == 0 && M == 0 && Rm == 11111)

```
BRAAZ <Xn>
```

Key A, register modifier (Z == 1 && M == 0)

```
BRAA <Xn>, <Xm|SP>
```

Key B, zero modifier (Z == 0 && M == 1 && Rm == 11111)

```
BRABZ <Xn>
```

Key B, register modifier (Z == 1 && M == 1)

```
BRAB <Xn>, <Xm|SP>
```

```
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean use_key_a = (M == '0');
boolean source_is_sp = ((Z == '1') && (m == 31));

if !HavePACExt() then
    UNDEFINED;

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

Assembler Symbols

<Xn>	Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.
<Xm|SP>	Is the 64-bit name of the general-purpose source register or stack pointer holding the modifier, encoded in the "Rm" field.

Operation

```
bits(64) target = X[n, 64];

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

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

// Value in BTypeNext will be used to set PSTATE.BTYPE
if InGuardedPage then
    if n == 16 || n == 17 then
        BTypeNext = '01';
    else
        BTypeNext = '11';
else
    BTypeNext = '01';
BranchTo(target, BranchType_INDIR, FALSE);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

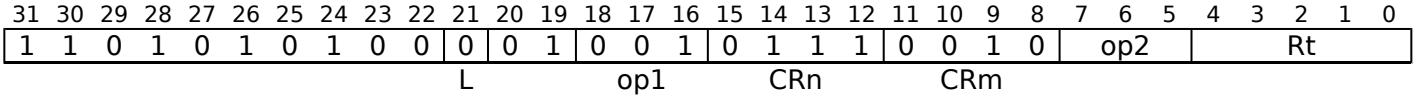
BRB

Branch Record Buffer. For more information, see *op0==0b01, cache maintenance, TLB maintenance, and address translation instructions*.

This is an alias of SYS. This means:

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

System
(FEAT_BRBE)



BRB <brb_op>{, <Xt>}

is equivalent to

SYS #1, C7, C2, #<op2>{, <Xt>}

and is the preferred disassembly when `SysOp('001','0111','0010',op2) == Sys_BRB`.

Assembler Symbols

<brb_op> Is a BRB instruction name, as listed for the BRB system instruction group, encoded in “op2”:

op2	<brb_op>	Architectural Feature
100	IALL	FEAT_BRBE
101	INJ	FEAT_BRBE

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

<Xt> Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

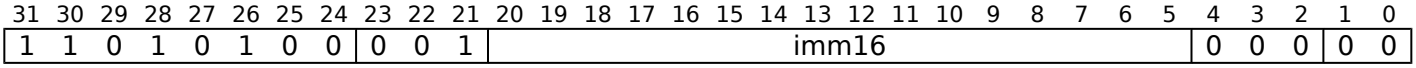
Operation

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BRK

Breakpoint instruction. A BRK instruction generates a Breakpoint Instruction exception. The PE records the exception in *ESR_ELx*, using the EC value 0x3c, and captures the value of the immediate argument in *ESR_ELx*.ISS.



BRK #<imm>

```
if HaveBTIExt() then
    SetBTypeCompatible(TRUE);
```

Assembler Symbols

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

Operation

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BTI

Branch Target Identification. A BTI instruction is used to guard against the execution of instructions which are not the intended target of a branch.

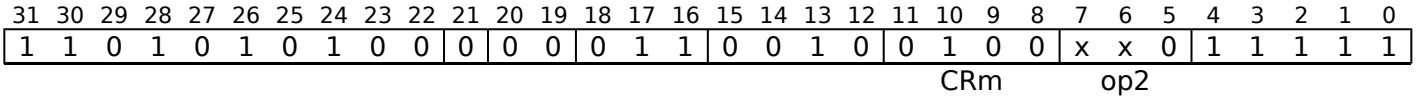
Outside of a guarded memory region, a BTI instruction executes as a NOP. Within a guarded memory region while *PSTATE*.BTYPE != 0b00, a BTI instruction compatible with the current value of PSTATE.BTYPE will not generate a Branch Target Exception and will allow execution of subsequent instructions within the memory region.

The operand <targets> passed to a BTI instruction determines the values of *PSTATE*.BTYPE which the BTI instruction is compatible with.

Note

Within a guarded memory region, when *PSTATE*.BTYPE != 0b00, all instructions will generate a Branch Target Exception, other than BRK, BTI, HLT, PACIASP, and PACIBSP, which might not. See the individual instructions for more information.

System
(FEAT_BTI)



BTI {<targets>}

```
SystemHintOp op;

if CRm:op2 == '0100 xx0' then
    op = SystemHintOp_BTI;
    // Check branch target compatibility between BTI instruction and PSTATE.BTYPE
    SetBTypeCompatible(BTypeCompatible_BTI(op2<2:1>));
else
    EndOfInstruction();
```

Assembler Symbols

<targets> Is the type of indirection, encoded in “op2<2:1>”:

op2<2:1>	<targets>
00	(omitted)
01	c
10	j
11	jc

Operation

```
case op of
    when SystemHintOp_YIELD
        Hint_Yield();

    when SystemHintOp_DGH
        Hint_DGH();

    when SystemHintOp_WFE
        integer localtimeout = 1 << 64;    // No local timeout event is generated
        Hint_WFE(localtimeout, WFxType_WFE);

    when SystemHintOp_WFI
        integer localtimeout = 1 << 64;    // No local timeout event is generated
        Hint_WFI(localtimeout, WFxType_WFI);

    when SystemHintOp_SEV
        SendEvent();

    when SystemHintOp_SEVL
        SendEventLocal();

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

    when SystemHintOp_PSB
        ProfilingSynchronizationBarrier();

    when SystemHintOp_TSB
        TraceSynchronizationBarrier();

    when SystemHintOp_CSDB
        ConsumptionOfSpeculativeDataBarrier();

    when SystemHintOp_BTI
        SetBTypeNext('00');

    otherwise    // do nothing
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CAS, CASA, CASAL, CASL

Compare and Swap word or doubleword in memory reads a 32-bit word or 64-bit doubleword from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- `CASA` and `CASAL` load from memory with acquire semantics.
- `CASL` and `CASAL` store to memory with release semantics.
- `CAS` has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

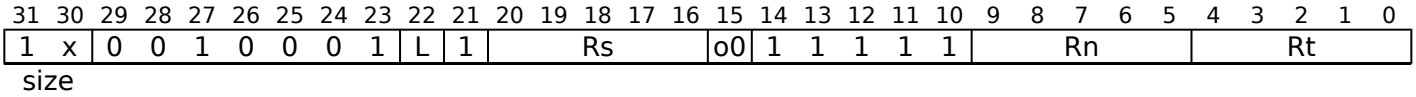
For information about memory accesses see *Load/Store addressing modes*.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is <Ws>, or <Xs>, is restored to the value held in the register before the instruction was executed.

No offset

(FEAT_LSE)



32-bit CAS (size == 10 && L == 0 && o0 == 0)

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

32-bit CASA (size == 10 && L == 1 && o0 == 0)

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

32-bit CASAL (size == 10 && L == 1 && o0 == 1)

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

32-bit CASL (size == 10 && L == 0 && o0 == 1)

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

64-bit CAS (size == 11 && L == 0 && o0 == 0)

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

64-bit CASA (size == 11 && L == 1 && o0 == 0)

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

64-bit CASAL (size == 11 && L == 1 && o0 == 1)

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

64-bit CASL (size == 11 && L == 0 && o0 == 1)

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

```
if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if L == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

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

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

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

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

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

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CASB, CASAB, CASALB, CASLB

Compare and Swap byte in memory reads an 8-bit byte from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- CASAB and CASALB load from memory with acquire semantics.
- CASLB and CASALB store to memory with release semantics.
- CASB has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

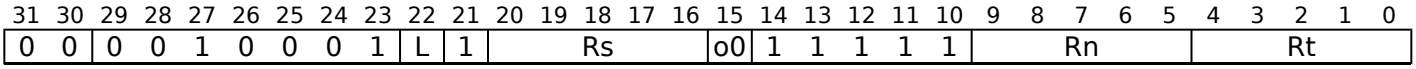
For information about memory accesses see *Load/Store addressing modes*.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is <Ws>, is restored to the values held in the register before the instruction was executed.

No offset

(FEAT_LSE)



CASAB (L == 1 && o0 == 0)

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

CASALB (L == 1 && o0 == 1)

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

CASB (L == 0 && o0 == 0)

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

CASLB (L == 0 && o0 == 1)

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

```
if !HaveAtomicExt() then UNDEFINED;

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

AccType ldacctype = if L == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) comparevalue;
bits(8) newvalue;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

comparevalue = X[s, 8];
newvalue = X[t, 8];

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

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

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CASH, CASAH, CASALH, CASLH

Compare and Swap halfword in memory reads a 16-bit halfword from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

- CASAH and CASALH load from memory with acquire semantics.
- CASLH and CASALH store to memory with release semantics.
- CAS has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

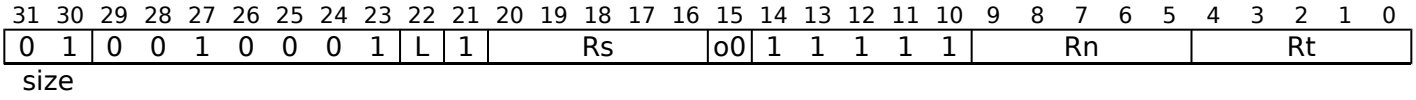
For information about memory accesses see *Load/Store addressing modes*.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is <Ws>, is restored to the values held in the register before the instruction was executed.

No offset

(FEAT_LSE)



CASAH (L == 1 && o0 == 0)

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

CASALH (L == 1 && o0 == 1)

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

CASH (L == 0 && o0 == 0)

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

CASLH (L == 0 && o0 == 1)

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

```
if !HaveAtomicExt() then UNDEFINED;

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

AccType ldacctype = if L == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if o0 == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be conditionally stored, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) comparevalue;
bits(16) newvalue;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

comparevalue = X[s, 16];
newvalue = X[t, 16];

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

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

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CASP, CASPA, CASPAL, CASPL

Compare and Swap Pair of words or doublewords in memory reads a pair of 32-bit words or 64-bit doublewords from memory, and compares them against the values held in the first pair of registers. If the comparison is equal, the values in the second pair of registers are written to memory. If the writes are performed, the reads and writes occur atomically such that no other modification of the memory location can take place between the reads and writes.

- `CASPA` and `CASPAL` load from memory with acquire semantics.
- `CASPL` and `CASPAL` store to memory with release semantics.
- `CAS` has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

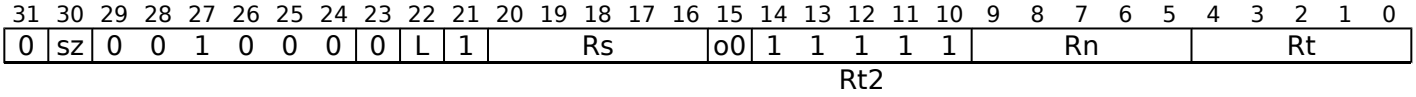
For information about memory accesses see *Load/Store addressing modes*.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the registers which are compared and loaded, that is <Ws> and <W(s+1)>, or <Xs> and <X(s+1)>, are restored to the values held in the registers before the instruction was executed.

No offset

(FEAT_LSE)



32-bit CASP (sz == 0 && L == 0 && o0 == 0)

```
CASP <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{,#0}]
```

32-bit CASPA (sz == 0 && L == 1 && o0 == 0)

```
CASPA <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{,#0}]
```

32-bit CASPAL (sz == 0 && L == 1 && o0 == 1)

```
CASPAL <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{,#0}]
```

32-bit CASPL (sz == 0 && L == 0 && o0 == 1)

```
CASPL <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{,#0}]
```

64-bit CASP (sz == 1 && L == 0 && o0 == 0)

```
CASP <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}]
```

64-bit CASPA (sz == 1 && L == 1 && o0 == 0)

```
CASPA <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}]
```

64-bit CASPAL (sz == 1 && L == 1 && o0 == 1)

```
CASPAL <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}]
```

64-bit CASPL (sz == 1 && L == 0 && o0 == 1)

```
CASPL <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}]
```

```
if !HaveAtomicExt() then UNDEFINED;
if Rs<0> == '1' then UNDEFINED;
if Rt<0> == '1' then UNDEFINED;

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

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

Assembler Symbols

<Ws>	Is the 32-bit name of the first general-purpose register to be compared and loaded, encoded in the "Rs" field. <Ws> must be an even-numbered register.
<W(s+1)>	Is the 32-bit name of the second general-purpose register to be compared and loaded.
<Wt>	Is the 32-bit name of the first general-purpose register to be conditionally stored, encoded in the "Rt" field. <Wt> must be an even-numbered register.
<W(t+1)>	Is the 32-bit name of the second general-purpose register to be conditionally stored.
<Xs>	Is the 64-bit name of the first general-purpose register to be compared and loaded, encoded in the "Rs" field. <Xs> must be an even-numbered register.
<X(s+1)>	Is the 64-bit name of the second general-purpose register to be compared and loaded.
<Xt>	Is the 64-bit name of the first general-purpose register to be conditionally stored, encoded in the "Rt" field. <Xt> must be an even-numbered register.

<X(t+1)> Is the 64-bit name of the second general-purpose register to be conditionally stored.

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

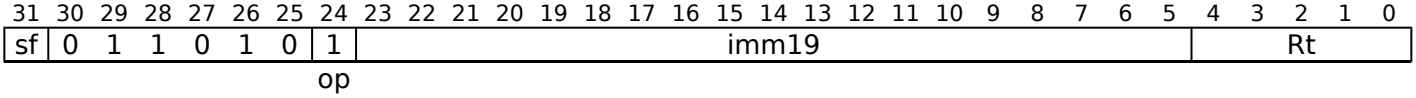
Operation

```
bits(64) address;
bits(2*datasize) comparevalue;
bits(2*datasize) newvalue;
bits(2*datasize) data;

bits(datasize) s1 = X[s, datasize];
bits(datasize) s2 = X[s+1, datasize];
bits(datasize) t1 = X[t, datasize];
bits(datasize) t2 = X[t+1, datasize];
comparevalue = if BigEndian(ldacctype) then s1:s2 else s2:s1;
newvalue = if BigEndian(stacctype) then t1:t2 else t2:t1;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

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

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CBNZ

Compare and Branch on Nonzero compares the value in a register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is not equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect the condition flags.



32-bit (sf == 0)

```
CBNZ <Wt>, <label>
```

64-bit (sf == 1)

```
CBNZ <Xt>, <label>
```

```
integer t = UInt(Rt);
integer datasize = if sf == '1' then 64 else 32;
bits(64) offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
<label>	Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

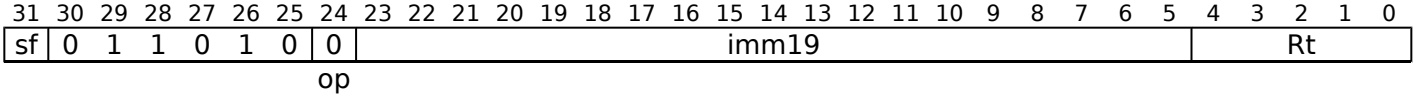
Operation

```
bits(datasize) operand1 = X[t, datasize];
if IsZero(operand1) == FALSE then
    BranchTo(PC[] + offset, BranchType_DIR, TRUE);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CBZ

Compare and Branch on Zero compares the value in a register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.



32-bit (sf == 0)

```
CBZ <Wt>, <label>
```

64-bit (sf == 1)

```
CBZ <Xt>, <label>
```

```
integer t = UInt(Rt);
integer datasize = if sf == '1' then 64 else 32;
bits(64) offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be tested, encoded in the "Rt" field.
<label>	Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

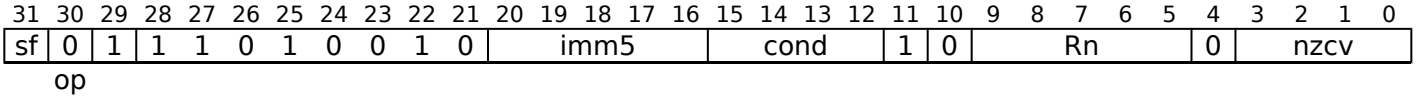
Operation

```
bits(datasize) operand1 = X[t, datasize];
if IsZero(operand1) == TRUE then
    BranchTo(PC[] + offset, BranchType_DIR, TRUE);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CCMN (immediate)

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



32-bit (sf == 0)

```
CCMN <Wn>, #<imm>, #<nzcv>, <cond>
```

64-bit (sf == 1)

```
CCMN <Xn>, #<imm>, #<nzcv>, <cond>
```

```
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(4) flags = nzcv;
bits(datasize) imm = ZeroExtend(imm5, datasize);
```

Assembler Symbols

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

Operation

```
if ConditionHolds(cond) then
    bits(datasize) operand1 = X[n, datasize];
    (-, flags) = AddWithCarry(operand1, imm, '0');
PSTATE.<N,Z,C,V> = flags;
```

Operational information

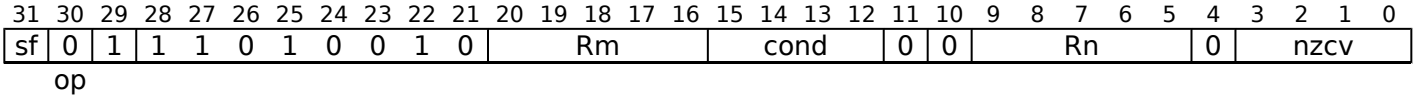
If PSTATE.DIT is 1:

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CCMN (register)

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



32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

Assembler Symbols

<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<nzcv>	Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
<cond>	Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

```
if ConditionHolds(cond) then
    bits(datasize) operand1 = X[n, datasize];
    bits(datasize) operand2 = X[m, datasize];
    (-, flags) = AddWithCarry(operand1, operand2, '0');
PSTATE.<N,Z,C,V> = flags;
```

Operational information

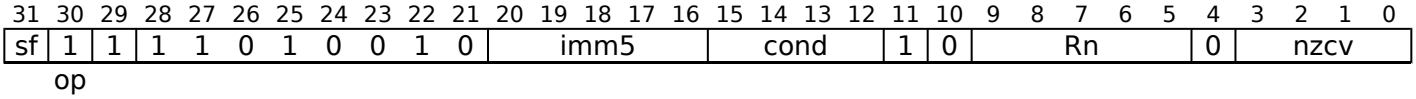
If PSTATE.DIT is 1:

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CCMP (immediate)

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



32-bit (sf == 0)

```
CCMP <Wn>, #<imm>, #<nzcv>, <cond>
```

64-bit (sf == 1)

```
CCMP <Xn>, #<imm>, #<nzcv>, <cond>
```

```
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(4) flags = nzcv;
bits(datasize) imm = ZeroExtend(imm5, datasize);
```

Assembler Symbols

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

Operation

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

Operational information

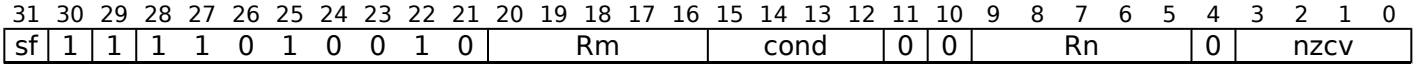
If PSTATE.DIT is 1:

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CCMP (register)

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



32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

Assembler Symbols

<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<nzcv>	Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
<cond>	Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

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

Operational information

If PSTATE.DIT is 1:

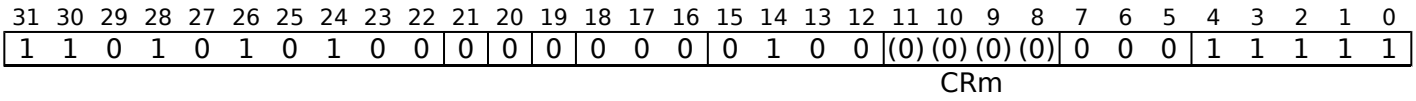
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CFINV

Invert Carry Flag. This instruction inverts the value of the PSTATE.C flag.

System
(FEAT_FlagM)



CFINV

```
if !HaveFlagManipulateExt() then UNDEFINED;
```

Operation

```
PSTATE.C = NOT(PSTATE.C);
```

Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

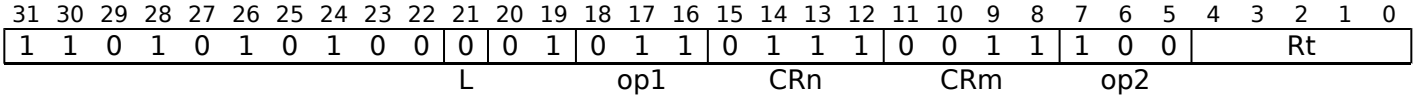
CFP

Control Flow Prediction Restriction by Context prevents control flow predictions that predict execution addresses based on information gathered from earlier execution within a particular execution context. Control flow predictions determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot be used to exploitatively control speculative execution occurring after the instruction is complete and synchronized.
For more information, see *CFP RCTX, Control Flow Prediction Restriction by Context*.

This is an alias of SYS. This means:

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

System
(FEAT_SPECRES)



CFP RCTX, <Xt>

is equivalent to

SYS #3, C7, C3, #4, <Xt>

and is always the preferred disassembly.

Assembler Symbols

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

Operation

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

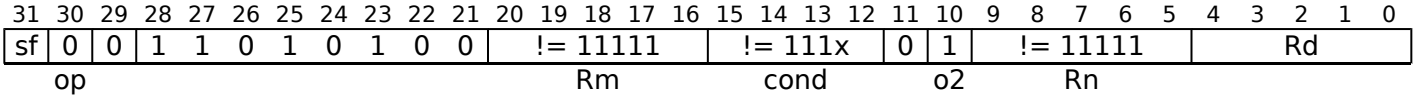
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CINC

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

This is an alias of CSINC. This means:

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



32-bit (sf == 0)

```
CINC <Wd>, <Wn>, <cond>
```

is equivalent to

```
CSINC <Wd>, <Wn>, <Wn>, invert(<cond>)
```

and is the preferred disassembly when `Rn == Rm`.

64-bit (sf == 1)

```
CINC <Xd>, <Xn>, <cond>
```

is equivalent to

```
CSINC <Xd>, <Xn>, <Xn>, invert(<cond>)
```

and is the preferred disassembly when `Rn == Rm`.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
<cond>	Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

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

Operational information

If PSTATE.DIT is 1:

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

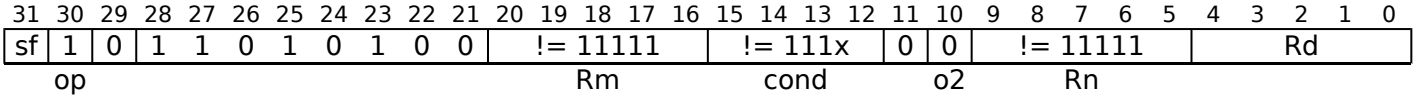
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CINV

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

This is an alias of CSINV. This means:

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



32-bit (sf == 0)

```
CINV <Wd>, <Wn>, <cond>
```

is equivalent to

```
CSINV <Wd>, <Wn>, <Wn>, invert(<cond>)
```

and is the preferred disassembly when `Rn == Rm`.

64-bit (sf == 1)

```
CINV <Xd>, <Xn>, <cond>
```

is equivalent to

```
CSINV <Xd>, <Xn>, <Xn>, invert(<cond>)
```

and is the preferred disassembly when `Rn == Rm`.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
<cond>	Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

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

Operational information

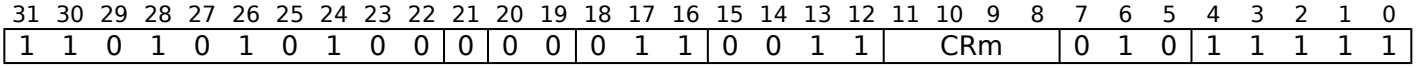
If PSTATE.DIT is 1:

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CLREX

Clear Exclusive clears the local monitor of the executing PE.




```
CLREX {#<imm>}

// CRm field is ignored
```

Assembler Symbols

<imm>	Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15 and encoded in the "CRm" field.

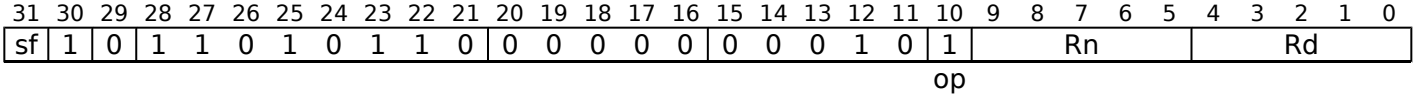
Operation

```
ClearExclusiveLocal(ProcessorID());
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CLS

Count Leading Sign bits counts the number of leading bits of the source register that have the same value as the most significant bit of the register, and writes the result to the destination register. This count does not include the most significant bit of the source register.



32-bit (sf == 0)

```
CLS <Wd>, <Wn>
```

64-bit (sf == 1)

```
CLS <Xd>, <Xn>
```

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

Assembler Symbols

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

Operation

```
integer result;
bits(datasize) operand1 = X[n, datasize];

result = CountLeadingSignBits(operand1);

X[d, datasize] = result<datasize-1:0>;
```

Operational information

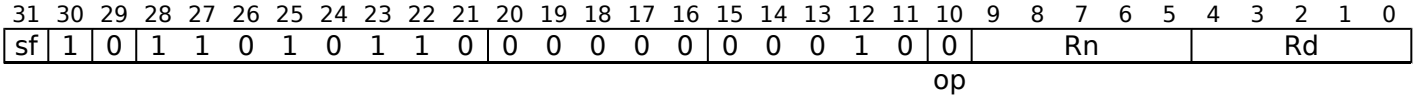
If PSTATE.DIT is 1:

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CLZ

Count Leading Zeros counts the number of binary zero bits before the first binary one bit in the value of the source register, and writes the result to the destination register.



32-bit (sf == 0)

```
CLZ <Wd>, <Wn>
```

64-bit (sf == 1)

```
CLZ <Xd>, <Xn>
```

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

Assembler Symbols

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

Operation

```
integer result;
bits(datasize) operand1 = X[n, datasize];

result = CountLeadingZeroBits(operand1);
X[d, datasize] = result<datasize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

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

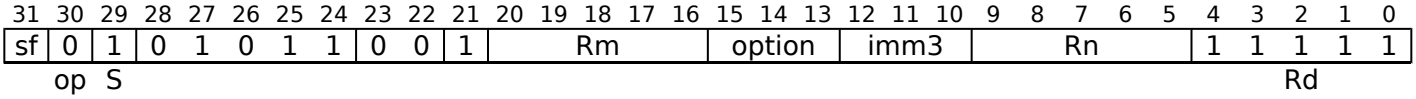
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CMN (extended register)

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

This is an alias of ADDS (extended register). This means:

- The encodings in this description are named to match the encodings of ADDS (extended register).
- The description of ADDS (extended register) gives the operational pseudocode for this instruction.



32-bit (sf == 0)

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

is equivalent to

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

and is always the preferred disassembly.

64-bit (sf == 1)

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

is equivalent to

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

and is always the preferred disassembly.

Assembler Symbols

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

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

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

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

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

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

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

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

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

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

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

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

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

Operation

The description of ADDS (extended register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

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

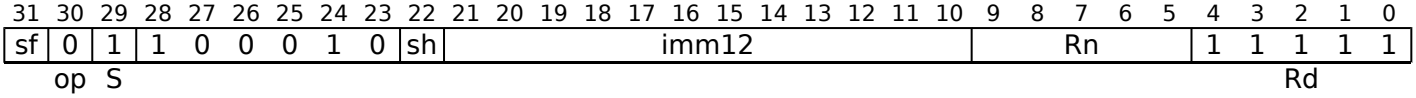
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CMN (immediate)

Compare Negative (immediate) adds a register value and an optionally-shifted immediate value. It updates the condition flags based on the result, and discards the result.

This is an alias of ADDS (immediate). This means:

- The encodings in this description are named to match the encodings of ADDS (immediate).
- The description of ADDS (immediate) gives the operational pseudocode for this instruction.



32-bit (sf == 0)

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

is equivalent to

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

and is always the preferred disassembly.

64-bit (sf == 1)

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

is equivalent to

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

and is always the preferred disassembly.

Assembler Symbols

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

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

Operation

The description of ADDS (immediate) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

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

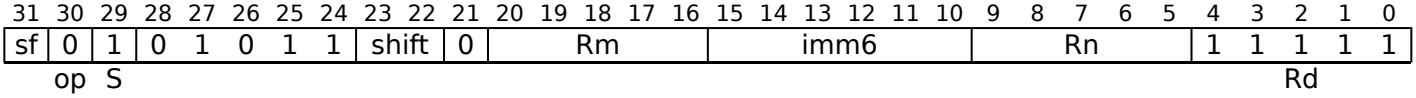
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CMN (shifted register)

Compare Negative (shifted register) adds a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

This is an alias of ADDS (shifted register). This means:

- The encodings in this description are named to match the encodings of ADDS (shifted register).
- The description of ADDS (shifted register) gives the operational pseudocode for this instruction.



32-bit (sf == 0)

```
CMN <Wn>, <Wm>{, <shift> #<amount>}
```

is equivalent to

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

and is always the preferred disassembly.

64-bit (sf == 1)

```
CMN <Xn>, <Xm>{, <shift> #<amount>}
```

is equivalent to

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

and is always the preferred disassembly.

Assembler Symbols

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

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

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

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

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

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

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

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

Operation

The description of ADDS (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

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

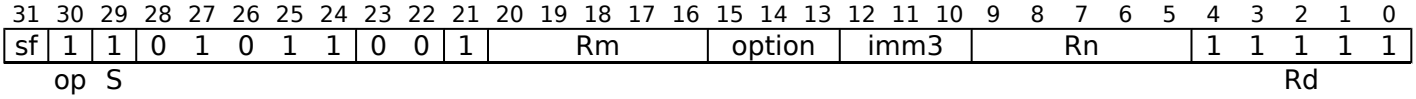
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CMP (extended register)

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

This is an alias of SUBS (extended register). This means:

- The encodings in this description are named to match the encodings of SUBS (extended register).
- The description of SUBS (extended register) gives the operational pseudocode for this instruction.



32-bit (sf == 0)

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

is equivalent to

```
SUBS WZR, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

and is always the preferred disassembly.

64-bit (sf == 1)

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

is equivalent to

```
SUBS XZR, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

and is always the preferred disassembly.

Assembler Symbols

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

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

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

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

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

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

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

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

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

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

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

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

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

Operation

The description of SUBS (extended register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

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

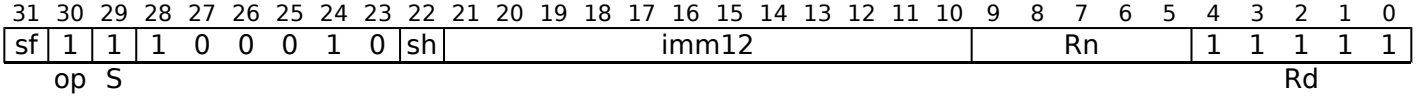
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CMP (immediate)

Compare (immediate) subtracts an optionally-shifted immediate value from a register value. It updates the condition flags based on the result, and discards the result.

This is an alias of SUBS (immediate). This means:

- The encodings in this description are named to match the encodings of SUBS (immediate).
- The description of SUBS (immediate) gives the operational pseudocode for this instruction.



32-bit (sf == 0)

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

is equivalent to

```
SUBS WZR, <Wn|WSP>, #<imm> {, <shift>}
```

and is always the preferred disassembly.

64-bit (sf == 1)

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

is equivalent to

```
SUBS XZR, <Xn|SP>, #<imm> {, <shift>}
```

and is always the preferred disassembly.

Assembler Symbols

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

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

Operation

The description of SUBS (immediate) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

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

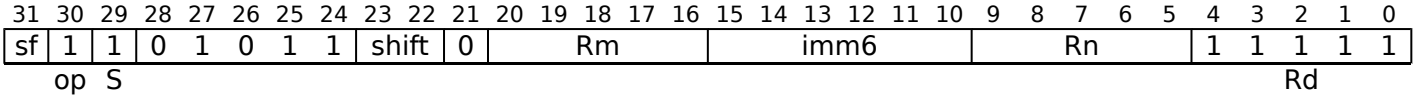
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CMP (shifted register)

Compare (shifted register) subtracts an optionally-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

This is an alias of SUBS (shifted register). This means:

- The encodings in this description are named to match the encodings of SUBS (shifted register).
- The description of SUBS (shifted register) gives the operational pseudocode for this instruction.



32-bit (sf == 0)

```
CMP <Wn>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
SUBS WZR, <Wn>, <Wm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

64-bit (sf == 1)

```
CMP <Xn>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
SUBS XZR, <Xn>, <Xm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler Symbols

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

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

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

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

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

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

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

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

Operation

The description of SUBS (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

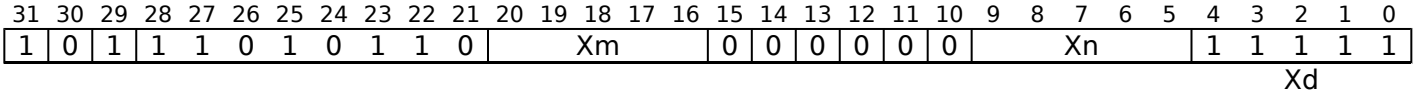
CMPP

Compare with Tag subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, updates the condition flags based on the result of the subtraction, and discards the result.

This is an alias of SUBPS. This means:

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

Integer
(FEAT_MTE)



CMPP <Xn|SP>, <Xm|SP>

is equivalent to

SUBPS XZR, <Xn|SP>, <Xm|SP>

and is always the preferred disassembly.

Assembler Symbols

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

Operation

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

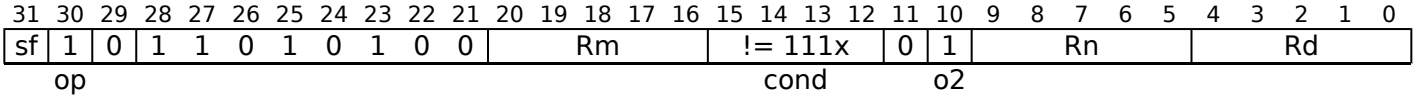
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CNEG

Conditional Negate returns, in the destination register, the negated value of the source register if the condition is TRUE, and otherwise returns the value of the source register.

This is an alias of CSNEG. This means:

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



32-bit (sf == 0)

```
CNEG <Wd>, <Wn>, <cond>
```

is equivalent to

```
CSNEG <Wd>, <Wn>, <Wn>, invert(<cond>)
```

and is the preferred disassembly when `Rn == Rm`.

64-bit (sf == 1)

```
CNEG <Xd>, <Xn>, <cond>
```

is equivalent to

```
CSNEG <Xd>, <Xn>, <Xn>, invert(<cond>)
```

and is the preferred disassembly when `Rn == Rm`.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
<cond>	Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

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

Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

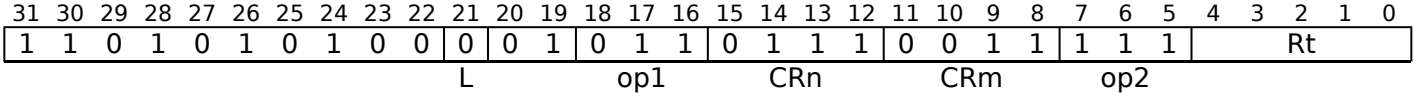
CPP

Cache Prefetch Prediction Restriction by Context prevents cache allocation predictions that predict execution addresses based on information gathered from earlier execution within a particular execution context. The actions of code in the target execution context or contexts appearing in program order before the instruction cannot exploitatively control cache prefetch predictions occurring after the instruction is complete and synchronized.
For more information, see *CPP RCTX, Cache Prefetch Prediction Restriction by Context*.

This is an alias of SYS. This means:

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

System
(FEAT_SPECRES)



CPP RCTX, <Xt>

is equivalent to

SYS #3, C7, C3, #7, <Xt>

and is always the preferred disassembly.

Assembler Symbols

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

Operation

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CPYFP, CPYFM, CPYFE

Memory Copy Forward-only. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFP, then CPYFM, and then CPYFE.

CPYFP performs some preconditioning of the arguments suitable for using the CPYFM instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFM performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFE performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFP, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFP, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFM, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFM, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFE, option A (encoded by PSTATE.C = 0), the format of the arguments is:

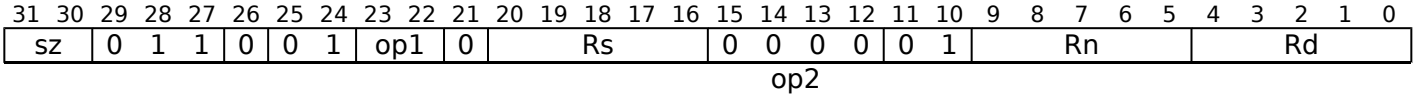
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFE, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.

- the value of Xd is written back with the lowest address that has not been copied to.

Integer
(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYFE [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYFM [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYFP [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if supports_option_a then
        nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to this instruction are valid for the epilogue.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
```

```
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
        Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CPYFPN, CPYFMN, CPYFEN

Memory Copy Forward-only, reads and writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPN, then CPYFMN, and then CPYFEN.

CPYFPN performs some preconditioning of the arguments suitable for using the CPYFMN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFEN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFEN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

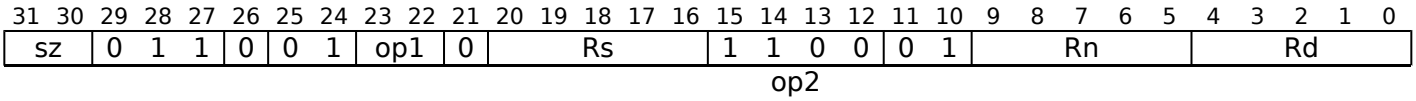
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFEN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with 0.

- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

Integer
(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYFEN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYFMN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYFPN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if supports_option_a then
        nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to this instruction are valid for the epilogue.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
```

```
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
        Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CPYFPRN, CPYFMRN, CPYFERN

Memory Copy Forward-only, reads non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRN, then CPYFMRN, and then CPYFERN.

CPYFPRN performs some preconditioning of the arguments suitable for using the CPYFMRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFERN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPRN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPRN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFERN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

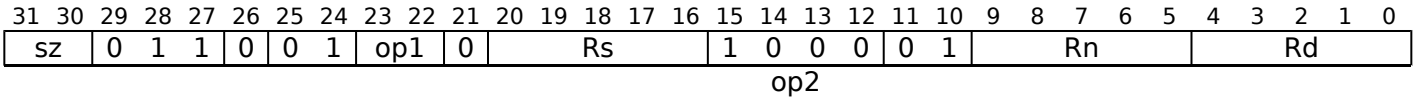
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFERN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with 0.

- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

Integer
(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYFERN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYFMRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYFPRN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if supports_option_a then
        nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, optior
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, optior

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to this instruction are valid for the epilogue.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
```

```
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
        Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CPYFPRT, CPYFMRT, CPYFERT

Memory Copy Forward-only, reads unprivileged. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRT, then CPYFMRT, and then CPYFERT.

CPYFPRT performs some preconditioning of the arguments suitable for using the CPYFMRT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMRT performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFERT performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPRT, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPRT, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMRT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMRT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFERT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

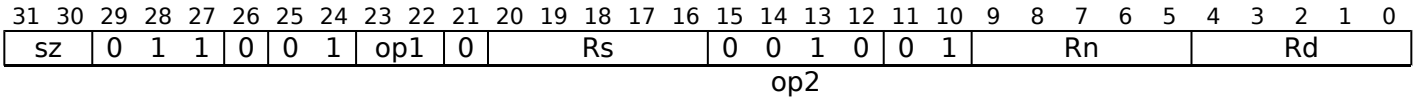
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFERT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with 0.

- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

Integer
(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYFERT [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYFMRT [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYFPRT [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if supports_option_a then
        nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, optior
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, optior

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to this instruction are valid for the epilogue.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
```

```
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
        Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CPYFPRTN, CPYFMRTN, CPYFERTN

Memory Copy Forward-only, reads unprivileged, reads and writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRTN, then CPYFMRTN, and then CPYFERTN.

CPYFPRTN performs some preconditioning of the arguments suitable for using the CPYFMRTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMRTN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFERTN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPRTN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPRTN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMRTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMRTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFERTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

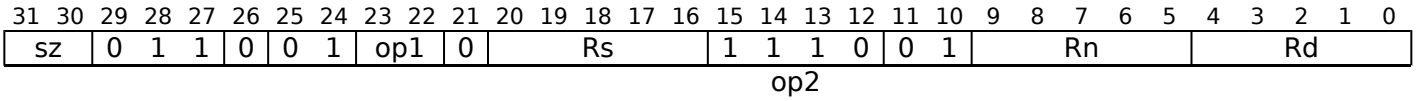
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFERTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with 0.

- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

Integer
(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYFERTN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYFMRTN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYFPRTN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if supports_option_a then
        nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, optior
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, optior

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to this instruction are valid for the epilogue.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
```

```
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
        Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

CPYFPRTRN, CPYFMRTRN, CPYFERTRN

Memory Copy Forward-only, reads unprivileged and non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRTRN, then CPYFMRTRN, and then CPYFERTRN.

CPYFPRTRN performs some preconditioning of the arguments suitable for using the CPYFMRTRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMRTRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFERTRN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPRTRN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPRTRN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMRTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMRTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFERTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

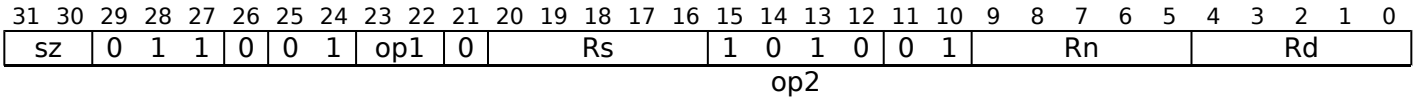
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFERTRN option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with 0.

- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

Integer
(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYFERTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYFMRTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYFPRTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if supports_option_a then
        nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, optior
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, optior

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to this instruction are valid for the epilogue.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
```

```
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
        Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

CPYFPRTWN, CPYFMRTWN, CPYFERTWN

Memory Copy Forward-only, reads unprivileged, writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPRTWN, then CPYFMRTWN, and then CPYFERTWN.

CPYFPRTWN performs some preconditioning of the arguments suitable for using the CPYFMRTWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMRTWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFERTWN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPRTWN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPRTWN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMRTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMRTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFERTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

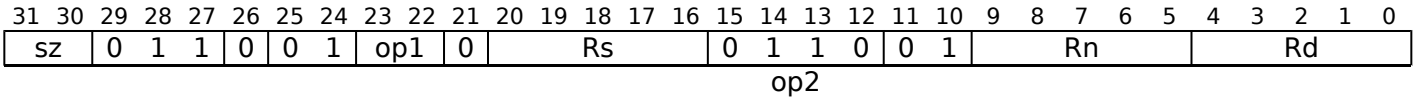
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFERTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with 0.

- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

Integer
(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYFERTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYFMRTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYFPRTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if supports_option_a then
        nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to this instruction are valid for the epilogue.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
```

```
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
        Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

CPYFPT, CPYFMT, CPYFET

Memory Copy Forward-only, reads and writes unprivileged. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPT, then CPYFMT, and then CPYFET.

CPYFPT performs some preconditioning of the arguments suitable for using the CPYFMT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMT performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFET performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPT, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPT, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFET, option A (encoded by PSTATE.C = 0), the format of the arguments is:

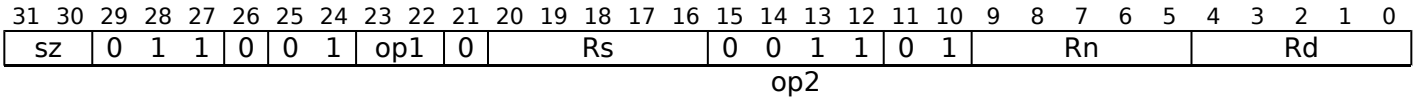
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFET, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with 0.

- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

Integer
(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYFET [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYFMT [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYFPT [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if supports_option_a then
        nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to this instruction are valid for the epilogue.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
```

```
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
        Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CPYFPTN, CPYFMTN, CPYFETN

Memory Copy Forward-only, reads and writes unprivileged and non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPTN, then CPYFMTN, and then CPYFETN.

CPYFPTN performs some preconditioning of the arguments suitable for using the CPYFMTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMTN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFETN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPTN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPTN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFETN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

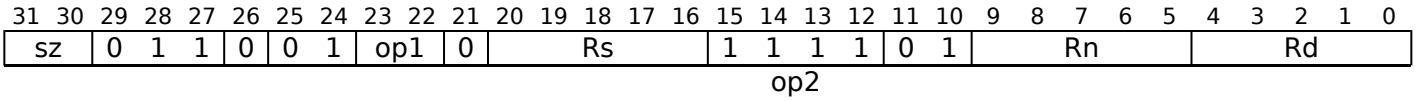
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFETN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with 0.

- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

Integer
(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYFETN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYFMTN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYFPTN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if supports_option_a then
        nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, optior
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, optior

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to this instruction are valid for the epilogue.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
```

```
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
        Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CPYFPTRN, CPYFMTRN, CPYFETRN

Memory Copy Forward-only, reads and writes unprivileged, reads non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPTRN, then CPYFMTRN, and then CPYFETRN.

CPYFPTRN performs some preconditioning of the arguments suitable for using the CPYFMTRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMTRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFETRN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPTRN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPTRN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFETRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

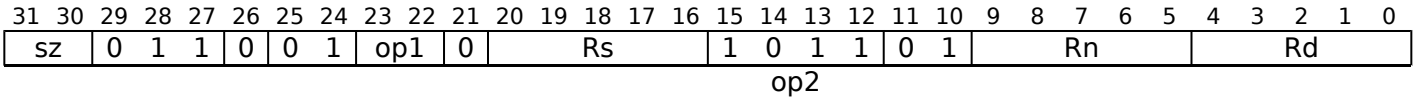
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFETRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with 0.

- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

Integer
(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYFETRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYFMTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYFPTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if supports_option_a then
        nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, optior
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, optior

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to this instruction are valid for the epilogue.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
```

```
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
        Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

CPYFPTWN, CPYFMTWN, CPYFETWN

Memory Copy Forward-only, reads and writes unprivileged, writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPTWN, then CPYFMTWN, and then CPYFETWN.

CPYFPTWN performs some preconditioning of the arguments suitable for using the CPYFMTWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMTWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFETWN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPTWN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPTWN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFETWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

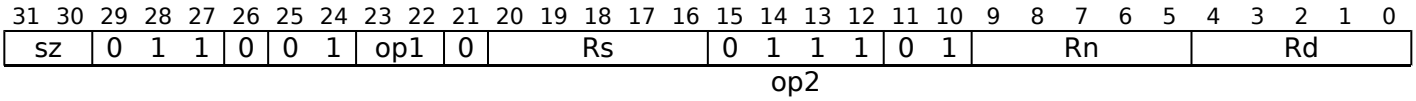
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFETWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with 0.

- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

Integer
(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYFETWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYFMTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYFPTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;
```

Assembler Symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.

<Xs> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if supports_option_a then
        nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, optior
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, optior

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to this instruction are valid for the epilogue.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
```

```
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
        Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CPYFPWN, CPYFMWN, CPYFEWN

Memory Copy Forward-only, writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWN, then CPYFMWN, and then CPYFEWN.

CPYFPWN performs some preconditioning of the arguments suitable for using the CPYFMWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFEWN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPWN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPWN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFEWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

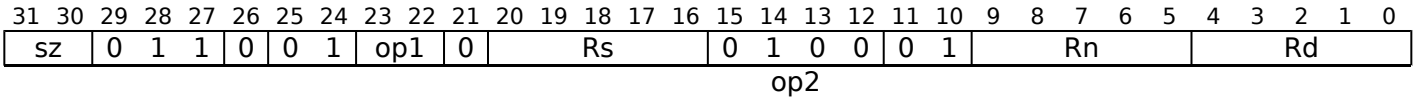
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFEWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with 0.

- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

Integer
(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYFEWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYFMWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYFPWN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if supports_option_a then
        nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, optior
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, optior

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to this instruction are valid for the epilogue.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
```

```
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
        Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

CPYFPWT, CPYFMWT, CPYFEWT

Memory Copy Forward-only, writes unprivileged. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWT, then CPYFMWT, and then CPYFEWT.

CPYFPWT performs some preconditioning of the arguments suitable for using the CPYFMWT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMWT performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFEWT performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPWT, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPWT, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMWT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMWT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFEWT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

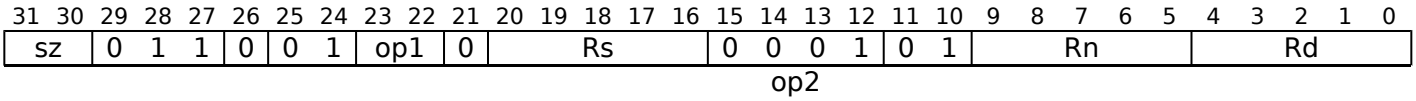
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFEWT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with 0.

- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

Integer
(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYFEWT [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYFMWT [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYFPWT [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if supports_option_a then
        nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, optior
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, optior

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to this instruction are valid for the epilogue.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
```

```
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
        Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CPYFPWTN, CPYFMWTN, CPYFEWTN

Memory Copy Forward-only, writes unprivileged, reads and writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWTN, then CPYFMWTN, and then CPYFEWTN.

CPYFPWTN performs some preconditioning of the arguments suitable for using the CPYFMWTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMWTN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFEWTN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPWTN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPWTN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMWTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMWTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFEWTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

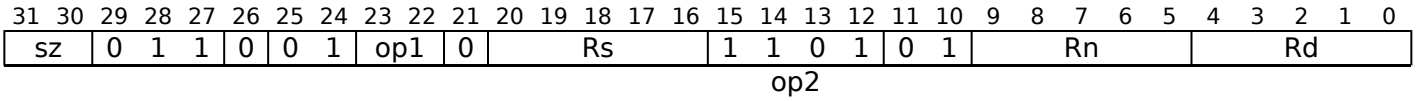
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFEWTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with 0.

- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

Integer
(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYFEWTN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYFMWTN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYFPWTN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if supports_option_a then
        nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, optior
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, optior

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to this instruction are valid for the epilogue.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
```

```
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
        Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CPYFPWTRN, CPYFMWTRN, CPYFEWTRN

Memory Copy Forward-only, writes unprivileged, reads non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWTRN, then CPYFMWTRN, and then CPYFEWTRN.

CPYFPWTRN performs some preconditioning of the arguments suitable for using the CPYFMWTRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMWTRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFEWTRN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPWTRN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPWTRN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMWTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMWTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFEWTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

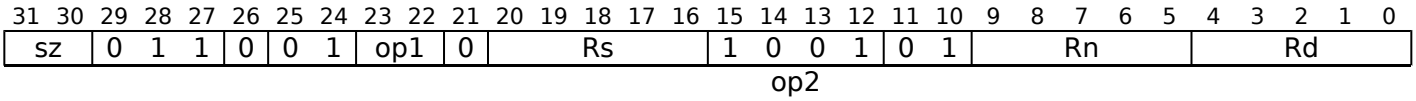
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFEWTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with 0.

- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

Integer
(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYFEWTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYFMWTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYFPWTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if supports_option_a then
        nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, optior
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, optior

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to this instruction are valid for the epilogue.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
```

```
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
        Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

CPYFPWTWN, CPYFMWTWN, CPYFEWTWN

Memory Copy Forward-only, writes unprivileged and non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYFPWTWN, then CPYFMWTWN, and then CPYFEWTWN.

CPYFPWTWN performs some preconditioning of the arguments suitable for using the CPYFMWTWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFMWTWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYFEWTWN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

The memory copy performed by these instructions is in the forward direction only, so the instructions are suitable for a memory copy only where there is no overlap between the source and destination locations, or where the source address is greater than the destination address.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYFPWTWN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + saturated Xn.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of CPYFPWTWN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For CPYFMWTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.

For CPYFMWTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.

For CPYFEWTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

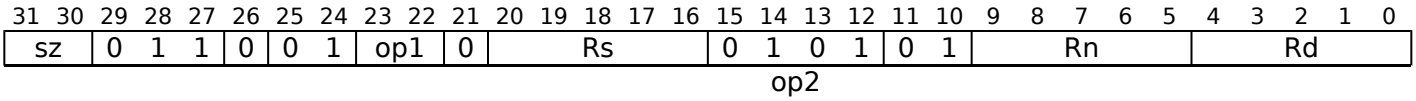
- Xn is treated as a signed 64-bit number and holds -1* the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from -Xn.
- Xd holds the lowest address that the copy is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For CPYFEWTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be copied in the memory copy in total.
- Xs holds the lowest address that the copy is copied from.
- Xd holds the lowest address that the copy is copied to.
- At the end of the instruction:
    - the value of Xn is written back with 0.

- the value of Xs is written back with the lowest address that has not been copied from.
- the value of Xd is written back with the lowest address that has not been copied to.

Integer
(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYFEWTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYFMWTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYFPWTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

if d == s || s == n || d == n then UNDEFINED;
if d == 31 || s == 31 || n == 31 then UNDEFINED;
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
CheckMOPSEnabled();

integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63> == '1' then cpysize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if supports_option_a then
        nzcv = '0000';
        // Copy in the forward direction offsets the arguments.
        toaddress = toaddress + cpysize;
        fromaddress = fromaddress + cpysize;
        cpysize = Zeros(64) - cpysize;
    else
        nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, optior
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, optior

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to this instruction are valid for the epilogue.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
```

```
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= -1 * SInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
        Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
        cpysize = cpysize + B;
        stagecpysize = stagecpysize + B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
        Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
        fromaddress = fromaddress + B;
        toaddress = toaddress + B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CPYP, CPYM, CPYE

Memory Copy. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYP, then CPYM, and then CPYE.

CPYP performs some preconditioning of the arguments suitable for using the CPYM instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYM performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYE performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYP, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYP, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYP, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYM, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is copied to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYM, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:

        - Xs holds the lowest address that the copy is copied from.
        - Xd holds the lowest address that the copy is copied to.
        - At the end of the instruction:
            - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
            - the value of Xs is written back with the lowest address that has not been copied from.
            - the value of Xd is written back with the lowest address that has not been copied to.
    - If the copy is in the backward direction (PSTATE.N == 1), then:
        - Xs holds the highest address that the copy is copied from +1.
        - Xd holds the highest address that the copy is copied to +1.
        - At the end of the instruction:
            - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
            - the value of Xs is written back with the highest address that has not been copied from +1.
            - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYE, option A (encoded by PSTATE.C = 0), the format of the arguments is:

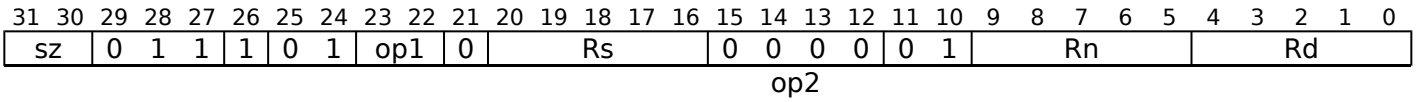
- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
    - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
    - Xs holds the lowest address that the copy is copied from.
    - Xd holds the lowest address that the copy is made to.
    - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
    - Xn holds the number of bytes remaining to be copied in the memory copy in total.
    - Xs holds the highest address that the copy is copied from -Xn+1.
    - Xd holds the highest address that the copy is copied to -Xn+1.
    - At the end of the instruction, the value of Xn is written back with 0.

For CPYE, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
    - Xs holds the lowest address that the copy is copied from.
    - Xd holds the lowest address that the copy is copied to.
    - At the end of the instruction:
        - the value of Xn is written back with 0.
        - the value of Xs is written back with the lowest address that has not been copied from.
        - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
    - Xs holds the highest address that the copy is copied from.
    - Xd holds the highest address that the copy is copied to.
    - At the end of the instruction:
        - the value of Xn is written back with 0.
        - the value of Xs is written back with the highest address that has not been copied from +1.
        - the value of Xd is written back with the highest address that has not been copied to +1.

Integer

(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYE [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYM [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYP [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
    c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFF<63:0>;

    boolean forward;
    if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) < UInt(toaddress<55
        forward = TRUE;
    elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>)
        forward = FALSE;
    else
        forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

    if supports_option_a then
        nzcv = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
    else
        if !forward then
            // Copy in the reverse direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            nzcv = '1010';
        else
            nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
```

```
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to the epilogue are valid.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);

            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;
            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        if nzcv<3> == '0' then // PSTATE.N
            readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
            Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress + B;
            toaddress = toaddress + B;
        else
            readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
            Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress - B;
            toaddress = toaddress - B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

CPYPN, CPYMN, CPYEN

Memory Copy, reads and writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPN, then CPYMN, and then CPYEN.

CPYPN performs some preconditioning of the arguments suitable for using the CPYMN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYEN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPN, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.

- If the copy is in the forward direction (PSTATE.N == 0), then:
    - Xs holds the lowest address that the copy is copied from.
    - Xd holds the lowest address that the copy is copied to.
    - At the end of the instruction:
        - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
        - the value of Xs is written back with the lowest address that has not been copied from.
        - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
    - Xs holds the highest address that the copy is copied from +1.
    - Xd holds the highest address that the copy is copied to +1.
    - At the end of the instruction:
        - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
        - the value of Xs is written back with the highest address that has not been copied from +1.
        - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYEN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

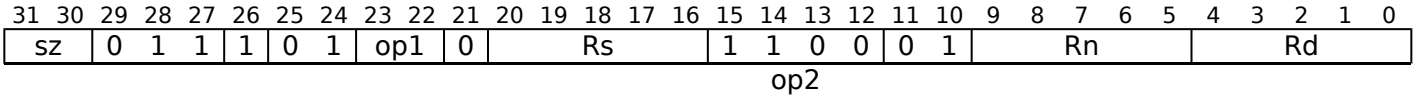
- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
    - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
    - Xs holds the lowest address that the copy is copied from -Xn.
    - Xd holds the lowest address that the copy is made to -Xn.
    - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
    - Xn holds the number of bytes remaining to be copied in the memory copy in total.
    - Xs holds the highest address that the copy is copied from -Xn+1.
    - Xd holds the highest address that the copy is copied to -Xn+1.
    - At the end of the instruction, the value of Xn is written back with 0.

For CPYEN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
    - Xs holds the lowest address that the copy is copied from.
    - Xd holds the lowest address that the copy is copied to.
    - At the end of the instruction:
        - the value of Xn is written back with 0.
        - the value of Xs is written back with the lowest address that has not been copied from.
        - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
    - Xs holds the highest address that the copy is copied from +1.
    - Xd holds the highest address that the copy is copied to +1.
    - At the end of the instruction:
        - the value of Xn is written back with 0.
        - the value of Xs is written back with the highest address that has not been copied from +1.
        - the value of Xd is written back with the highest address that has not been copied to +1.

Integer

(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYEN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYMN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYPN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
    c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFF<63:0>;

    boolean forward;
    if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) < UInt(toaddress<55
        forward = TRUE;
    elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>)
        forward = FALSE;
    else
        forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

    if supports_option_a then
        nzcv = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
    else
        if !forward then
            // Copy in the reverse direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            nzcv = '1010';
        else
            nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
```

```
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to the epilogue are valid.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);

            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;
            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        if nzcv<3> == '0' then // PSTATE.N
            readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
            Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress + B;
            toaddress = toaddress + B;
        else
            readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
            Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress - B;
            toaddress = toaddress - B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

CPYPRN, CPYMRN, CPYERN

Memory Copy, reads non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRN, then CPYMRN, and then CPYERN.

CPYPRN performs some preconditioning of the arguments suitable for using the CPYMRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYERN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPRN, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPRN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPRN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.

- If the copy is in the forward direction (PSTATE.N == 0), then:
    - Xs holds the lowest address that the copy is copied from.
    - Xd holds the lowest address that the copy is copied to.
    - At the end of the instruction:
        - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
        - the value of Xs is written back with the lowest address that has not been copied from.
        - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
    - Xs holds the highest address that the copy is copied from +1.
    - Xd holds the highest address that the copy is copied to +1.
    - At the end of the instruction:
        - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
        - the value of Xs is written back with the highest address that has not been copied from +1.
        - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYERN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

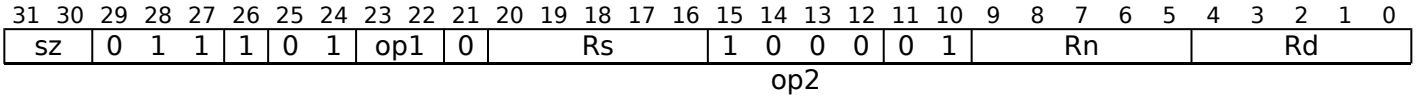
- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
    - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
    - Xs holds the lowest address that the copy is copied from -Xn.
    - Xd holds the lowest address that the copy is made to -Xn.
    - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
    - Xn holds the number of bytes remaining to be copied in the memory copy in total.
    - Xs holds the highest address that the copy is copied from -Xn+1.
    - Xd holds the highest address that the copy is copied to -Xn+1.
    - At the end of the instruction, the value of Xn is written back with 0.

For CPYERN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
    - Xs holds the lowest address that the copy is copied from.
    - Xd holds the lowest address that the copy is copied to.
    - At the end of the instruction:
        - the value of Xn is written back with 0.
        - the value of Xs is written back with the lowest address that has not been copied from.
        - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
    - Xs holds the highest address that the copy is copied from +1.
    - Xd holds the highest address that the copy is copied to +1.
    - At the end of the instruction:
        - the value of Xn is written back with 0.
        - the value of Xs is written back with the highest address that has not been copied from +1.
        - the value of Xd is written back with the highest address that has not been copied to +1.

Integer

(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYERN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYMRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYPRN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
    c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFF<63:0>;

    boolean forward;
    if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) < UInt(toaddress<55
        forward = TRUE;
    elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>)
        forward = FALSE;
    else
        forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

    if supports_option_a then
        nzcv = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
    else
        if !forward then
            // Copy in the reverse direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            nzcv = '1010';
        else
            nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
```

```
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to the epilogue are valid.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);

            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;
            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        if nzcv<3> == '0' then // PSTATE.N
            readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
            Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress + B;
            toaddress = toaddress + B;
        else
            readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
            Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress - B;
            toaddress = toaddress - B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

CPYPRT, CPYMRT, CPYERT

Memory Copy, reads unprivileged. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRT, then CPYMRT, and then CPYERT.

CPYPRT performs some preconditioning of the arguments suitable for using the CPYMRT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMRT performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYERT performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPRT, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPRT, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPRT, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMRT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMRT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.

- If the copy is in the forward direction (PSTATE.N == 0), then:
    - Xs holds the lowest address that the copy is copied from.
    - Xd holds the lowest address that the copy is copied to.
    - At the end of the instruction:
        - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
        - the value of Xs is written back with the lowest address that has not been copied from.
        - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
    - Xs holds the highest address that the copy is copied from +1.
    - Xd holds the highest address that the copy is copied to +1.
    - At the end of the instruction:
        - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
        - the value of Xs is written back with the highest address that has not been copied from +1.
        - the value of Xd is written back with the highest address that has not been copied to +1.

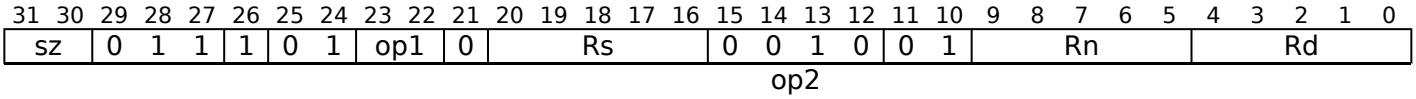
For CPYERT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
    - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
    - Xs holds the lowest address that the copy is copied from -Xn.
    - Xd holds the lowest address that the copy is made to -Xn.
    - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
    - Xn holds the number of bytes remaining to be copied in the memory copy in total.
    - Xs holds the highest address that the copy is copied from -Xn+1.
    - Xd holds the highest address that the copy is copied to -Xn+1.
    - At the end of the instruction, the value of Xn is written back with 0.

For CPYERT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
    - Xs holds the lowest address that the copy is copied from.
    - Xd holds the lowest address that the copy is copied to.
    - At the end of the instruction:
        - the value of Xn is written back with 0.
        - the value of Xs is written back with the lowest address that has not been copied from.
        - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
    - Xs holds the highest address that the copy is copied from +1.
    - Xd holds the highest address that the copy is copied to +1.
    - At the end of the instruction:
        - the value of Xn is written back with 0.
        - the value of Xs is written back with the highest address that has not been copied from +1.
        - the value of Xd is written back with the highest address that has not been copied to +1.

Integer
(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYERT [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYMRT [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYPRT [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
    c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFF<63:0>;

    boolean forward;
    if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) < UInt(toaddress<55
        forward = TRUE;
    elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>)
        forward = FALSE;
    else
        forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

    if supports_option_a then
        nzcv = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
    else
        if !forward then
            // Copy in the reverse direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            nzcv = '1010';
        else
            nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
```

```
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to the epilogue are valid.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);

            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;
            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        if nzcv<3> == '0' then // PSTATE.N
            readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
            Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress + B;
            toaddress = toaddress + B;
        else
            readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
            Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress - B;
            toaddress = toaddress - B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

CPYPRTN, CPYMRTN, CPYERTN

Memory Copy, reads unprivileged, reads and writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRTN, then CPYMRTN, and then CPYERTN.

CPYPRTN performs some preconditioning of the arguments suitable for using the CPYMRTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMRTN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYERTN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPRTN, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPRTN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPRTN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMRTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMRTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.

- If the copy is in the forward direction (PSTATE.N == 0), then:
    - Xs holds the lowest address that the copy is copied from.
    - Xd holds the lowest address that the copy is copied to.
    - At the end of the instruction:
        - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
        - the value of Xs is written back with the lowest address that has not been copied from.
        - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
    - Xs holds the highest address that the copy is copied from +1.
    - Xd holds the highest address that the copy is copied to +1.
    - At the end of the instruction:
        - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
        - the value of Xs is written back with the highest address that has not been copied from +1.
        - the value of Xd is written back with the highest address that has not been copied to +1.

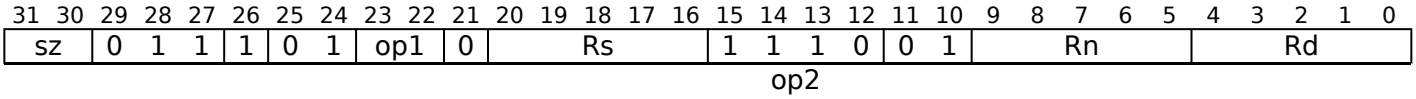
For CPYERTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
    - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
    - Xs holds the lowest address that the copy is copied from -Xn.
    - Xd holds the lowest address that the copy is made to -Xn.
    - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
    - Xn holds the number of bytes remaining to be copied in the memory copy in total.
    - Xs holds the highest address that the copy is copied from -Xn+1.
    - Xd holds the highest address that the copy is copied to -Xn+1.
    - At the end of the instruction, the value of Xn is written back with 0.

For CPYERTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
    - Xs holds the lowest address that the copy is copied from.
    - Xd holds the lowest address that the copy is copied to.
    - At the end of the instruction:
        - the value of Xn is written back with 0.
        - the value of Xs is written back with the lowest address that has not been copied from.
        - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
    - Xs holds the highest address that the copy is copied from +1.
    - Xd holds the highest address that the copy is copied to +1.
    - At the end of the instruction:
        - the value of Xn is written back with 0.
        - the value of Xs is written back with the highest address that has not been copied from +1.
        - the value of Xd is written back with the highest address that has not been copied to +1.

Integer
(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYERTN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYMRTN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYPRTN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
    c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFF<63:0>;

    boolean forward;
    if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) < UInt(toaddress<55
        forward = TRUE;
    elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>)
        forward = FALSE;
    else
        forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

    if supports_option_a then
        nzcv = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
    else
        if !forward then
            // Copy in the reverse direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            nzcv = '1010';
        else
            nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
```

```
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to the epilogue are valid.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);

            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;
            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        if nzcv<3> == '0' then // PSTATE.N
            readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
            Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress + B;
            toaddress = toaddress + B;
        else
            readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
            Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress - B;
            toaddress = toaddress - B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

CPYPRTRN, CPYMRTRN, CPYERTRN

Memory Copy, reads unprivileged and non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRTRN, then CPYMRTRN, and then CPYERTRN.

CPYPRTRN performs some preconditioning of the arguments suitable for using the CPYMRTRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMRTRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYERTRN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPRTRN, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPRTRN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPRTRN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMRTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMRTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.

- If the copy is in the forward direction (PSTATE.N == 0), then:
    - Xs holds the lowest address that the copy is copied from.
    - Xd holds the lowest address that the copy is copied to.
    - At the end of the instruction:
        - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
        - the value of Xs is written back with the lowest address that has not been copied from.
        - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
    - Xs holds the highest address that the copy is copied from +1.
    - Xd holds the highest address that the copy is copied to +1.
    - At the end of the instruction:
        - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
        - the value of Xs is written back with the highest address that has not been copied from +1.
        - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYERTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

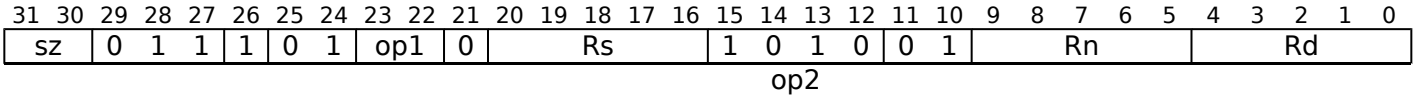
- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
    - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
    - Xs holds the lowest address that the copy is copied from -Xn.
    - Xd holds the lowest address that the copy is made to -Xn.
    - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
    - Xn holds the number of bytes remaining to be copied in the memory copy in total.
    - Xs holds the highest address that the copy is copied from -Xn+1.
    - Xd holds the highest address that the copy is copied to -Xn+1.
    - At the end of the instruction, the value of Xn is written back with 0.

For CPYERTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
    - Xs holds the lowest address that the copy is copied from.
    - Xd holds the lowest address that the copy is copied to.
    - At the end of the instruction:
        - the value of Xn is written back with 0.
        - the value of Xs is written back with the lowest address that has not been copied from.
        - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
    - Xs holds the highest address that the copy is copied from +1.
    - Xd holds the highest address that the copy is copied to +1.
    - At the end of the instruction:
        - the value of Xn is written back with 0.
        - the value of Xs is written back with the highest address that has not been copied from +1.
        - the value of Xd is written back with the highest address that has not been copied to +1.

Integer

(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYERTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYMRTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYPRTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
    c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFF<63:0>;

    boolean forward;
    if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) < UInt(toaddress<55
        forward = TRUE;
    elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>)
        forward = FALSE;
    else
        forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

    if supports_option_a then
        nzcv = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
    else
        if !forward then
            // Copy in the reverse direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            nzcv = '1010';
        else
            nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
```

```
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to the epilogue are valid.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);

            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;
            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        if nzcv<3> == '0' then // PSTATE.N
            readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
            Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress + B;
            toaddress = toaddress + B;
        else
            readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
            Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress - B;
            toaddress = toaddress - B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

CPYPRTWN, CPYMRTWN, CPYERTWN

Memory Copy, reads unprivileged, writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPRTWN, then CPYMRTWN, and then CPYERTWN.

CPYPRTWN performs some preconditioning of the arguments suitable for using the CPYMRTWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMRTWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYERTWN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPRTWN, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPRTWN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPRTWN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMRTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMRTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.

- If the copy is in the forward direction (PSTATE.N == 0), then:
    - Xs holds the lowest address that the copy is copied from.
    - Xd holds the lowest address that the copy is copied to.
    - At the end of the instruction:
        - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
        - the value of Xs is written back with the lowest address that has not been copied from.
        - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
    - Xs holds the highest address that the copy is copied from +1.
    - Xd holds the highest address that the copy is copied to +1.
    - At the end of the instruction:
        - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
        - the value of Xs is written back with the highest address that has not been copied from +1.
        - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYERTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

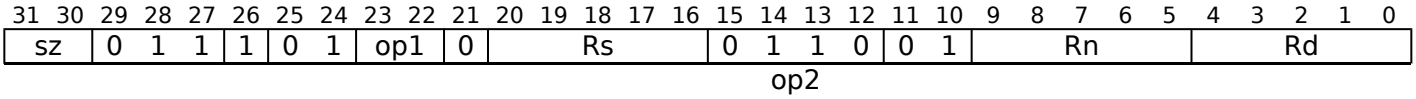
- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
    - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
    - Xs holds the lowest address that the copy is copied from -Xn.
    - Xd holds the lowest address that the copy is made to -Xn.
    - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
    - Xn holds the number of bytes remaining to be copied in the memory copy in total.
    - Xs holds the highest address that the copy is copied from -Xn+1.
    - Xd holds the highest address that the copy is copied to -Xn+1.
    - At the end of the instruction, the value of Xn is written back with 0.

For CPYERTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
    - Xs holds the lowest address that the copy is copied from.
    - Xd holds the lowest address that the copy is copied to.
    - At the end of the instruction:
        - the value of Xn is written back with 0.
        - the value of Xs is written back with the lowest address that has not been copied from.
        - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
    - Xs holds the highest address that the copy is copied from +1.
    - Xd holds the highest address that the copy is copied to +1.
    - At the end of the instruction:
        - the value of Xn is written back with 0.
        - the value of Xs is written back with the highest address that has not been copied from +1.
        - the value of Xd is written back with the highest address that has not been copied to +1.

Integer

(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYERTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYMRTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYPRTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
    c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFF<63:0>;

    boolean forward;
    if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) < UInt(toaddress<55
        forward = TRUE;
    elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>)
        forward = FALSE;
    else
        forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

    if supports_option_a then
        nzcv = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
    else
        if !forward then
            // Copy in the reverse direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            nzcv = '1010';
        else
            nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
```

```
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to the epilogue are valid.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);

            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;
            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        if nzcv<3> == '0' then // PSTATE.N
            readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
            Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress + B;
            toaddress = toaddress + B;
        else
            readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
            Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress - B;
            toaddress = toaddress - B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

CPYPT, CPYMT, CPYET

Memory Copy, reads and writes unprivileged. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPT, then CPYMT, and then CPYET.

CPYPT performs some preconditioning of the arguments suitable for using the CPYMT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMT performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYET performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPT, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPT, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPT, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.

- If the copy is in the forward direction (PSTATE.N == 0), then:
    - Xs holds the lowest address that the copy is copied from.
    - Xd holds the lowest address that the copy is copied to.
    - At the end of the instruction:
        - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
        - the value of Xs is written back with the lowest address that has not been copied from.
        - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
    - Xs holds the highest address that the copy is copied from +1.
    - Xd holds the highest address that the copy is copied to +1.
    - At the end of the instruction:
        - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
        - the value of Xs is written back with the highest address that has not been copied from +1.
        - the value of Xd is written back with the highest address that has not been copied to +1.

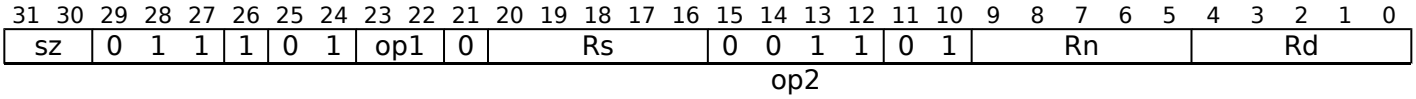
For CPYET, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
    - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
    - Xs holds the lowest address that the copy is copied from -Xn.
    - Xd holds the lowest address that the copy is made to -Xn.
    - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
    - Xn holds the number of bytes remaining to be copied in the memory copy in total.
    - Xs holds the highest address that the copy is copied from -Xn+1.
    - Xd holds the highest address that the copy is copied to -Xn+1.
    - At the end of the instruction, the value of Xn is written back with 0.

For CPYET, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
    - Xs holds the lowest address that the copy is copied from.
    - Xd holds the lowest address that the copy is copied to.
    - At the end of the instruction:
        - the value of Xn is written back with 0.
        - the value of Xs is written back with the lowest address that has not been copied from.
        - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
    - Xs holds the highest address that the copy is copied from +1.
    - Xd holds the highest address that the copy is copied to +1.
    - At the end of the instruction:
        - the value of Xn is written back with 0.
        - the value of Xs is written back with the highest address that has not been copied from +1.
        - the value of Xd is written back with the highest address that has not been copied to +1.

Integer
(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYET [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYMT [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYPT [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
    c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFF<63:0>;

    boolean forward;
    if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) < UInt(toaddress<55
        forward = TRUE;
    elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>)
        forward = FALSE;
    else
        forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

    if supports_option_a then
        nzcv = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
    else
        if !forward then
            // Copy in the reverse direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            nzcv = '1010';
        else
            nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
```

```
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to the epilogue are valid.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);

            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;
            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        if nzcv<3> == '0' then // PSTATE.N
            readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
            Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress + B;
            toaddress = toaddress + B;
        else
            readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
            Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress - B;
            toaddress = toaddress - B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

CPYPTN, CPYMTN, CPYETN

Memory Copy, reads and writes unprivileged and non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPTN, then CPYMTN, and then CPYETN.

CPYPTN performs some preconditioning of the arguments suitable for using the CPYMTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMTN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYETN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPTN, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPTN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPTN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.

- If the copy is in the forward direction (PSTATE.N == 0), then:
    - Xs holds the lowest address that the copy is copied from.
    - Xd holds the lowest address that the copy is copied to.
    - At the end of the instruction:
        - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
        - the value of Xs is written back with the lowest address that has not been copied from.
        - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
    - Xs holds the highest address that the copy is copied from +1.
    - Xd holds the highest address that the copy is copied to +1.
    - At the end of the instruction:
        - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
        - the value of Xs is written back with the highest address that has not been copied from +1.
        - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYETN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

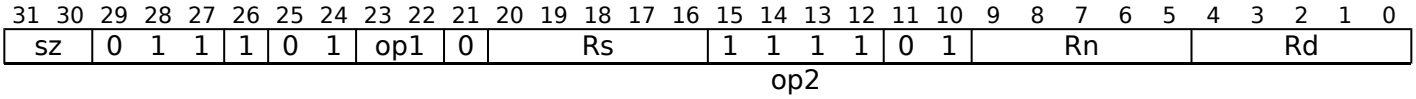
- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
    - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
    - Xs holds the lowest address that the copy is copied from -Xn.
    - Xd holds the lowest address that the copy is made to -Xn.
    - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
    - Xn holds the number of bytes remaining to be copied in the memory copy in total.
    - Xs holds the highest address that the copy is copied from -Xn+1.
    - Xd holds the highest address that the copy is copied to -Xn+1.
    - At the end of the instruction, the value of Xn is written back with 0.

For CPYETN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
    - Xs holds the lowest address that the copy is copied from.
    - Xd holds the lowest address that the copy is copied to.
    - At the end of the instruction:
        - the value of Xn is written back with 0.
        - the value of Xs is written back with the lowest address that has not been copied from.
        - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
    - Xs holds the highest address that the copy is copied from +1.
    - Xd holds the highest address that the copy is copied to +1.
    - At the end of the instruction:
        - the value of Xn is written back with 0.
        - the value of Xs is written back with the highest address that has not been copied from +1.
        - the value of Xd is written back with the highest address that has not been copied to +1.

Integer

(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYETN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYMTN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYPTN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
    c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFFF<63:0>;

    boolean forward;
    if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) < UInt(toaddress<55
        forward = TRUE;
    elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>)
        forward = FALSE;
    else
        forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

    if supports_option_a then
        nzcv = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
    else
        if !forward then
            // Copy in the reverse direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            nzcv = '1010';
        else
            nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
```

```
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to the epilogue are valid.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);

            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;
            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        if nzcv<3> == '0' then // PSTATE.N
            readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
            Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress + B;
            toaddress = toaddress + B;
        else
            readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
            Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress - B;
            toaddress = toaddress - B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

CPYPTRN, CPYMTRN, CPYETRN

Memory Copy, reads and writes unprivileged, reads non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPTRN, then CPYMTRN, and then CPYETRN.

CPYPTRN performs some preconditioning of the arguments suitable for using the CPYMTRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMTRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYETRN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPTRN, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPTRN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPTRN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.

- If the copy is in the forward direction (PSTATE.N == 0), then:
    - Xs holds the lowest address that the copy is copied from.
    - Xd holds the lowest address that the copy is copied to.
    - At the end of the instruction:
        - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
        - the value of Xs is written back with the lowest address that has not been copied from.
        - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
    - Xs holds the highest address that the copy is copied from +1.
    - Xd holds the highest address that the copy is copied to +1.
    - At the end of the instruction:
        - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
        - the value of Xs is written back with the highest address that has not been copied from +1.
        - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYETRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

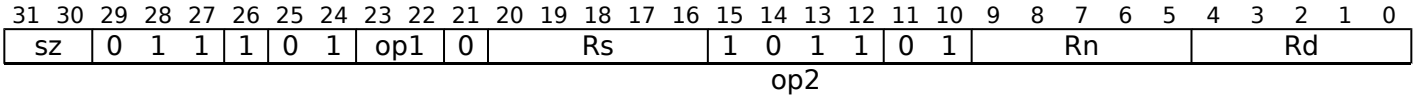
- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
    - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
    - Xs holds the lowest address that the copy is copied from -Xn.
    - Xd holds the lowest address that the copy is made to -Xn.
    - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
    - Xn holds the number of bytes remaining to be copied in the memory copy in total.
    - Xs holds the highest address that the copy is copied from -Xn+1.
    - Xd holds the highest address that the copy is copied to -Xn+1.
    - At the end of the instruction, the value of Xn is written back with 0.

For CPYETRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
    - Xs holds the lowest address that the copy is copied from.
    - Xd holds the lowest address that the copy is copied to.
    - At the end of the instruction:
        - the value of Xn is written back with 0.
        - the value of Xs is written back with the lowest address that has not been copied from.
        - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
    - Xs holds the highest address that the copy is copied from +1.
    - Xd holds the highest address that the copy is copied to +1.
    - At the end of the instruction:
        - the value of Xn is written back with 0.
        - the value of Xs is written back with the highest address that has not been copied from +1.
        - the value of Xd is written back with the highest address that has not been copied to +1.

Integer

(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYETRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYMTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYPTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
    c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFF<63:0>;

    boolean forward;
    if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) < UInt(toaddress<55
        forward = TRUE;
    elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>)
        forward = FALSE;
    else
        forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

    if supports_option_a then
        nzcv = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
    else
        if !forward then
            // Copy in the reverse direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            nzcv = '1010';
        else
            nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
```

```
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to the epilogue are valid.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);

            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;
            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        if nzcv<3> == '0' then // PSTATE.N
            readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
            Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress + B;
            toaddress = toaddress + B;
        else
            readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
            Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress - B;
            toaddress = toaddress - B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

CPYPTWN, CPYMTWN, CPYETWN

Memory Copy, reads and writes unprivileged, writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPTWN, then CPYMTWN, and then CPYETWN.

CPYPTWN performs some preconditioning of the arguments suitable for using the CPYMTWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMTWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYETWN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPTWN, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPTWN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPTWN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.

- If the copy is in the forward direction (PSTATE.N == 0), then:
    - Xs holds the lowest address that the copy is copied from.
    - Xd holds the lowest address that the copy is copied to.
    - At the end of the instruction:
        - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
        - the value of Xs is written back with the lowest address that has not been copied from.
        - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
    - Xs holds the highest address that the copy is copied from +1.
    - Xd holds the highest address that the copy is copied to +1.
    - At the end of the instruction:
        - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
        - the value of Xs is written back with the highest address that has not been copied from +1.
        - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYETWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

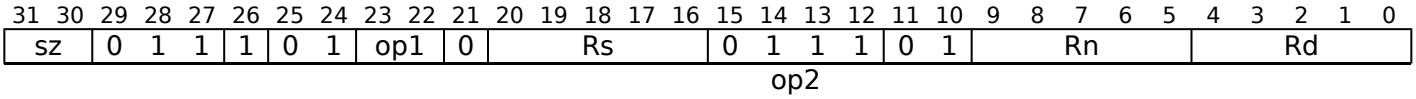
- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
    - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
    - Xs holds the lowest address that the copy is copied from -Xn.
    - Xd holds the lowest address that the copy is made to -Xn.
    - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
    - Xn holds the number of bytes remaining to be copied in the memory copy in total.
    - Xs holds the highest address that the copy is copied from -Xn+1.
    - Xd holds the highest address that the copy is copied to -Xn+1.
    - At the end of the instruction, the value of Xn is written back with 0.

For CPYETWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
    - Xs holds the lowest address that the copy is copied from.
    - Xd holds the lowest address that the copy is copied to.
    - At the end of the instruction:
        - the value of Xn is written back with 0.
        - the value of Xs is written back with the lowest address that has not been copied from.
        - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
    - Xs holds the highest address that the copy is copied from +1.
    - Xd holds the highest address that the copy is copied to +1.
    - At the end of the instruction:
        - the value of Xn is written back with 0.
        - the value of Xs is written back with the highest address that has not been copied from +1.
        - the value of Xd is written back with the highest address that has not been copied to +1.

Integer

(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYETWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYMTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYPTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
    c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFF<63:0>;

    boolean forward;
    if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) < UInt(toaddress<55
        forward = TRUE;
    elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>)
        forward = FALSE;
    else
        forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

    if supports_option_a then
        nzcv = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
    else
        if !forward then
            // Copy in the reverse direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            nzcv = '1010';
        else
            nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
```

```
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to the epilogue are valid.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);

            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;
            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        if nzcv<3> == '0' then // PSTATE.N
            readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
            Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress + B;
            toaddress = toaddress + B;
        else
            readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
            Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress - B;
            toaddress = toaddress - B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

CPYPWN, CPYMWN, CPYEWN

Memory Copy, writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWN, then CPYMWN, and then CPYEWN.

CPYPWN performs some preconditioning of the arguments suitable for using the CPYMWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYEWN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPWN, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPWN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPWN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.

- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYEWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

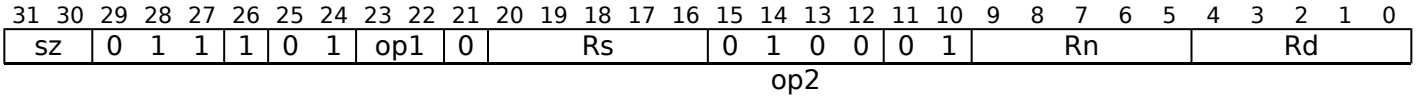
- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with 0.

For CPYEWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

Integer

(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYEWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYMWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYPWN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
    c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFF<63:0>;

    boolean forward;
    if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) < UInt(toaddress<55
        forward = TRUE;
    elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>)
        forward = FALSE;
    else
        forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

    if supports_option_a then
        nzcv = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
    else
        if !forward then
            // Copy in the reverse direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            nzcv = '1010';
        else
            nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
```

```
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to the epilogue are valid.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);

            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;
            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        if nzcv<3> == '0' then // PSTATE.N
            readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
            Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress + B;
            toaddress = toaddress + B;
        else
            readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
            Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress - B;
            toaddress = toaddress - B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

CPYPWT, CPYMWT, CPYEWT

Memory Copy, writes unprivileged. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWT, then CPYMWT, and then CPYEWT.

CPYPWT performs some preconditioning of the arguments suitable for using the CPYMWT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMWT performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYEWT performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPWT, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPWT, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPWT, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMWT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMWT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.

- If the copy is in the forward direction (PSTATE.N == 0), then:
    - Xs holds the lowest address that the copy is copied from.
    - Xd holds the lowest address that the copy is copied to.
    - At the end of the instruction:
        - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
        - the value of Xs is written back with the lowest address that has not been copied from.
        - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
    - Xs holds the highest address that the copy is copied from +1.
    - Xd holds the highest address that the copy is copied to +1.
    - At the end of the instruction:
        - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
        - the value of Xs is written back with the highest address that has not been copied from +1.
        - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYEWT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

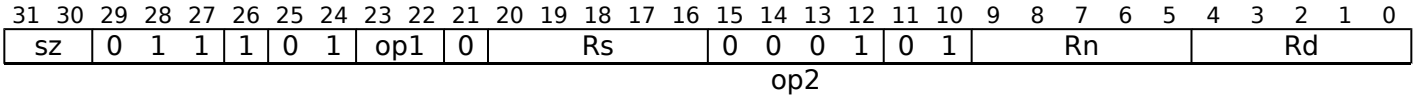
- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
    - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
    - Xs holds the lowest address that the copy is copied from -Xn.
    - Xd holds the lowest address that the copy is made to -Xn.
    - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
    - Xn holds the number of bytes remaining to be copied in the memory copy in total.
    - Xs holds the highest address that the copy is copied from -Xn+1.
    - Xd holds the highest address that the copy is copied to -Xn+1.
    - At the end of the instruction, the value of Xn is written back with 0.

For CPYEWT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
    - Xs holds the lowest address that the copy is copied from.
    - Xd holds the lowest address that the copy is copied to.
    - At the end of the instruction:
        - the value of Xn is written back with 0.
        - the value of Xs is written back with the lowest address that has not been copied from.
        - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
    - Xs holds the highest address that the copy is copied from +1.
    - Xd holds the highest address that the copy is copied to +1.
    - At the end of the instruction:
        - the value of Xn is written back with 0.
        - the value of Xs is written back with the highest address that has not been copied from +1.
        - the value of Xd is written back with the highest address that has not been copied to +1.

Integer

(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYEWT [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYMWT [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYPWT [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
    c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFF<63:0>;

    boolean forward;
    if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) < UInt(toaddress<55
        forward = TRUE;
    elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>)
        forward = FALSE;
    else
        forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

    if supports_option_a then
        nzcv = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
    else
        if !forward then
            // Copy in the reverse direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            nzcv = '1010';
        else
            nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
```

```
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to the epilogue are valid.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);

            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;
            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        if nzcv<3> == '0' then // PSTATE.N
            readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
            Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress + B;
            toaddress = toaddress + B;
        else
            readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
            Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress - B;
            toaddress = toaddress - B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

CPYPWTN, CPYMWTN, CPYEWTN

Memory Copy, writes unprivileged, reads and writes non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWTN, then CPYMWTN, and then CPYEWTN.

CPYPWTN performs some preconditioning of the arguments suitable for using the CPYMWTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMWTN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYEWTN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPWTN, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPWTN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPWTN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMWTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMWTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.

- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYEWTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

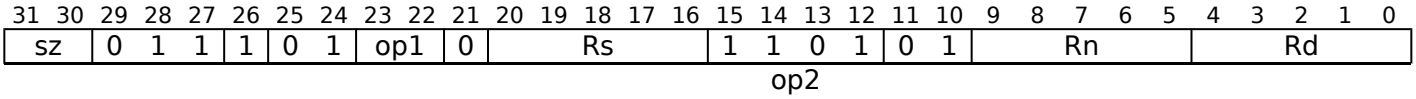
- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with 0.

For CPYEWTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

Integer

(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYEWTN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYMWTN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYPWTN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
    c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFF<63:0>;

    boolean forward;
    if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) < UInt(toaddress<55
        forward = TRUE;
    elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>)
        forward = FALSE;
    else
        forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

    if supports_option_a then
        nzcv = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
    else
        if !forward then
            // Copy in the reverse direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            nzcv = '1010';
        else
            nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
```

```
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to the epilogue are valid.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);

            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;
            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        if nzcv<3> == '0' then // PSTATE.N
            readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
            Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress + B;
            toaddress = toaddress + B;
        else
            readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
            Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress - B;
            toaddress = toaddress - B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

CPYPWTRN, CPYMWTRN, CPYEWTRN

Memory Copy, writes unprivileged, reads non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWTRN, then CPYMWTRN, and then CPYEWTRN.

CPYPWTRN performs some preconditioning of the arguments suitable for using the CPYMWTRN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMWTRN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYEWTRN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPWTRN, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPWTRN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPWTRN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMWTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMWTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.

- If the copy is in the forward direction (PSTATE.N == 0), then:
    - Xs holds the lowest address that the copy is copied from.
    - Xd holds the lowest address that the copy is copied to.
    - At the end of the instruction:
        - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
        - the value of Xs is written back with the lowest address that has not been copied from.
        - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
    - Xs holds the highest address that the copy is copied from +1.
    - Xd holds the highest address that the copy is copied to +1.
    - At the end of the instruction:
        - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
        - the value of Xs is written back with the highest address that has not been copied from +1.
        - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYEWTRN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

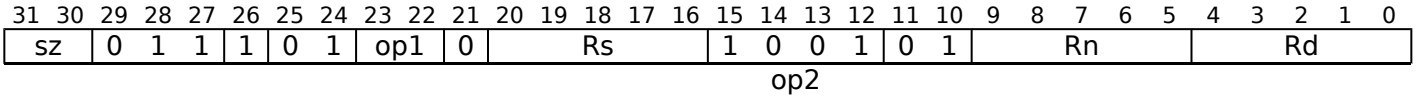
- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
    - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
    - Xs holds the lowest address that the copy is copied from -Xn.
    - Xd holds the lowest address that the copy is made to -Xn.
    - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
    - Xn holds the number of bytes remaining to be copied in the memory copy in total.
    - Xs holds the highest address that the copy is copied from -Xn+1.
    - Xd holds the highest address that the copy is copied to -Xn+1.
    - At the end of the instruction, the value of Xn is written back with 0.

For CPYEWTRN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
    - Xs holds the lowest address that the copy is copied from.
    - Xd holds the lowest address that the copy is copied to.
    - At the end of the instruction:
        - the value of Xn is written back with 0.
        - the value of Xs is written back with the lowest address that has not been copied from.
        - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
    - Xs holds the highest address that the copy is copied from +1.
    - Xd holds the highest address that the copy is copied to +1.
    - At the end of the instruction:
        - the value of Xn is written back with 0.
        - the value of Xs is written back with the highest address that has not been copied from +1.
        - the value of Xd is written back with the highest address that has not been copied to +1.

Integer

(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYEWTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYMWTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYPWTRN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
    c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFF<63:0>;

    boolean forward;
    if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) < UInt(toaddress<55
        forward = TRUE;
    elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>)
        forward = FALSE;
    else
        forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

    if supports_option_a then
        nzcv = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
    else
        if !forward then
            // Copy in the reverse direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            nzcv = '1010';
        else
            nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
```

```
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to the epilogue are valid.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);

            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;
            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        if nzcv<3> == '0' then // PSTATE.N
            readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
            Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress + B;
            toaddress = toaddress + B;
        else
            readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
            Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress - B;
            toaddress = toaddress - B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

CPYPWTWN, CPYMWTWN, CPYEWTWN

Memory Copy, writes unprivileged and non-temporal. These instructions perform a memory copy. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: CPYPWTWN, then CPYMWTWN, and then CPYEWTWN.

CPYPWTWN performs some preconditioning of the arguments suitable for using the CPYMWTWN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYMWTWN performs an IMPLEMENTATION DEFINED amount of the memory copy. CPYEWTWN performs the last part of the memory copy.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory copy allows some optimization of the size that can be performed.

For CPYPWTWN, the following saturation logic is applied:

If Xn<63:55> != 000000000, the copy size Xn is saturated to 0x007FFFFFFFFFFFF.

After that saturation logic is applied, the direction of the memory copy is based on the following algorithm:

If (Xs > Xd) && (Xd + saturated Xn) > Xs, then direction = forward

Elsif (Xs < Xd) && (Xs + saturated Xn) > Xd, then direction = backward

Else direction = IMPLEMENTATION DEFINED choice between forward and backward.

The architecture supports two algorithms for the memory copy: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of CPYPWTWN, option A (which results in encoding PSTATE.C = 0):

- PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the forward direction, then:
  - Xs holds the original Xs + saturated Xn.
  - Xd holds the original Xd + saturated Xn.
  - Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes copied.
- If the copy is in the backward direction, then:
  - Xs and Xd are unchanged.
  - Xn holds the saturated value of Xn - an IMPLEMENTATION DEFINED number of bytes copied.

After execution of CPYPWTWN, option B (which results in encoding PSTATE.C = 1):

- If the copy is in the forward direction, then:
  - Xs holds the original Xs + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {0,0,0}.
- If the copy is in the backward direction, then:
  - Xs holds the original Xs + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xd holds the original Xd + saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes copied.
  - PSTATE.{N,Z,V} are set to {1,0,0}.

For CPYMWTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be copied in the memory copy in total.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.

For CPYMWTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.

- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be copied in the memory copy in total.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

For CPYEWTWN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

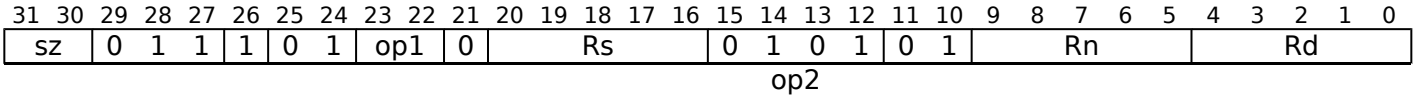
- Xn is treated as a signed 64-bit number.
- If the copy is in the forward direction (Xn is a negative number), then:
  - Xn holds -1* the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the lowest address that the copy is copied from -Xn.
  - Xd holds the lowest address that the copy is made to -Xn.
  - At the end of the instruction, the value of Xn is written back with 0.
- If the copy is in the backward direction (Xn is a positive number), then:
  - Xn holds the number of bytes remaining to be copied in the memory copy in total.
  - Xs holds the highest address that the copy is copied from -Xn+1.
  - Xd holds the highest address that the copy is copied to -Xn+1.
  - At the end of the instruction, the value of Xn is written back with 0.

For CPYEWTWN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes to be copied in the memory copy in total.
- If the copy is in the forward direction (PSTATE.N == 0), then:
  - Xs holds the lowest address that the copy is copied from.
  - Xd holds the lowest address that the copy is copied to.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the lowest address that has not been copied from.
    - the value of Xd is written back with the lowest address that has not been copied to.
- If the copy is in the backward direction (PSTATE.N == 1), then:
  - Xs holds the highest address that the copy is copied from +1.
  - Xd holds the highest address that the copy is copied to +1.
  - At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xs is written back with the highest address that has not been copied from +1.
    - the value of Xd is written back with the highest address that has not been copied to +1.

Integer

(FEAT_MOPS)



Epilogue (op1 == 10)

```
CPYEWTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Main (op1 == 01)

```
CPYMWTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

Prologue (op1 == 00)

```
CPYPWTWN [<Xd>]!, [<Xs>]!, <Xn>!
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(4) options = op2;

MOPSStage stage;
case op1 of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise SEE "Memory Copy and Memory Set";

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || s == 31 || n == 31 then
    c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xs>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the source address, encoded in the "Rs" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the source address and is updated by the instruction, encoded in the "Rs" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be transferred, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be transferred and is updated by the instruction to encode the remaining size and destination, encoded in the "Rn" field.

Operation

```
integer N = MaxBlockSizeCopiedBytes();
bits(64) toaddress = X[d, 64];
bits(64) fromaddress = X[s, 64];
bits(64) cpysize = X[n, 64];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagecpysize;
bits(8*N) readdata;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
(racctype, wacctype) = MemCpyAccessTypes(options);

if stage == MOPSStage_Prologue then
    if cpysize<63:55> != '000000000' then cpysize = 0x007FFFFFFFFFFF<63:0>;

    boolean forward;
    if ((UInt(fromaddress<55:0>) > UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0>) < UInt(toaddress<55
        forward = TRUE;
    elsif ((UInt(fromaddress<55:0>) < UInt(toaddress<55:0>)) && (UInt(fromaddress<55:0> + cpysize<55:0>)
        forward = FALSE;
    else
        forward = MemCpyDirectionChoice(fromaddress, toaddress, cpysize);

    if supports_option_a then
        nzcv = '0000';
        if forward then
            // Copy in the forward direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            cpysize = Zeros(64) - cpysize;
    else
        if !forward then
            // Copy in the reverse direction offsets the arguments.
            toaddress = toaddress + cpysize;
            fromaddress = fromaddress + cpysize;
            nzcv = '1010';
        else
            nzcv = '0010';

    // IMP DEF selection of the amount covered by pre-processing.
    stagecpysize = CPYPreSizeChoice(toaddress, fromaddress, cpysize);
    assert stagecpysize<63> == cpysize<63> || stagecpysize == Zeros(64);

    if SInt(cpysize) > 0 then
        assert SInt(stagecpysize) <= SInt(cpysize);
    else
        assert SInt(stagecpysize) >= SInt(cpysize);
else
    boolean zero_size_exceptions = MemCpyZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(cpysize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, option

    bits(64) postsize = CPYPostSizeChoice(toaddress, fromaddress, cpysize);
    assert postsize<63> == cpysize<63> || SInt(postsize) == 0;

    if stage == MOPSStage_Main then
```

```
        stagecpysize = cpysize - postsize;

        // Check if the parameters to this instruction are valid.
        if MemCpyParametersIllformedM(toaddress, fromaddress, cpysize) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);
    else
        stagecpysize = postsize;

        // Check if the parameters to the epilogue are valid.
        if (cpysize != postsize || MemCpyParametersIllformedE(toaddress, fromaddress, cpysize)) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = TRUE;
            MismatchedMemCpyException(supports_option_a, d, s, n, wrong_option, from_epilogue, options);

if supports_option_a then
    while SInt(stagecpysize) != 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);

        if SInt(cpysize) < 0 then
            assert B <= -1 * SInt(stagecpysize);

            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;
            cpysize = cpysize + B;
            stagecpysize = stagecpysize + B;
        else
            assert B <= SInt(stagecpysize);

            cpysize = cpysize - B;
            stagecpysize = stagecpysize - B;
            readdata<B*8-1:0> = Mem[fromaddress+cpysize, B, racctype];
            Mem[toaddress+cpysize, B, wacctype] = readdata<B*8-1:0>;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
else
    while UInt(stagecpysize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = CPYSizeChoice(toaddress, fromaddress, cpysize);
        assert B <= UInt(stagecpysize);

        if nzcv<3> == '0' then // PSTATE.N
            readdata<B*8-1:0> = Mem[fromaddress, B, racctype];
            Mem[toaddress, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress + B;
            toaddress = toaddress + B;
        else
            readdata<B*8-1:0> = Mem[fromaddress-B, B, racctype];
            Mem[toaddress-B, B, wacctype] = readdata<B*8-1:0>;
            fromaddress = fromaddress - B;
            toaddress = toaddress - B;

        cpysize = cpysize - B;
        stagecpysize = stagecpysize - B;

        if stage != MOPSStage_Prologue then
            X[n, 64] = cpysize;
            X[d, 64] = toaddress;
            X[s, 64] = fromaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = cpysize;
    X[d, 64] = toaddress;
    X[s, 64] = fromaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

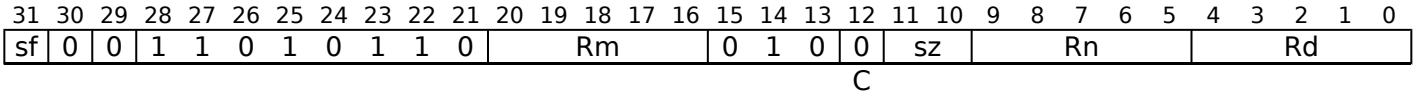
CRC32B, CRC32H, CRC32W, CRC32X

CRC32 checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x04C11DB7 is used for the CRC calculation.
In an Armv8.0 implementation, this is an OPTIONAL instruction. From Armv8.1, it is mandatory for all implementations to implement this instruction.

Note

ID_AA64ISAR0_EL1.CRC32 indicates whether this instruction is supported.

CRC
(FEAT_CRC32)



CRC32B (sf == 0 && sz == 00)

```
CRC32B <Wd>, <Wn>, <Wm>
```

CRC32H (sf == 0 && sz == 01)

```
CRC32H <Wd>, <Wn>, <Wm>
```

CRC32W (sf == 0 && sz == 10)

```
CRC32W <Wd>, <Wn>, <Wm>
```

CRC32X (sf == 1 && sz == 11)

```
CRC32X <Wd>, <Wn>, <Xm>
```

```
if !HaveCRCExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sf == '1' && sz != '11' then UNDEFINED;
if sf == '0' && sz == '11' then UNDEFINED;
integer size = 8 << UInt(sz);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose accumulator output register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose accumulator input register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose data source register, encoded in the "Rm" field.
<Wm>	Is the 32-bit name of the general-purpose data source register, encoded in the "Rm" field.

Operation

```
bits(32) acc = X[n, 32];    // accumulator
bits(size) val = X[m, size];    // input value
bits(32) poly = 0x04C11DB7<31:0>;

bits(32+size) tempacc = BitReverse(acc):Zeros(size);
bits(size+32) tempval = BitReverse(val):Zeros(32);

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
X[d, 32] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
```

Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

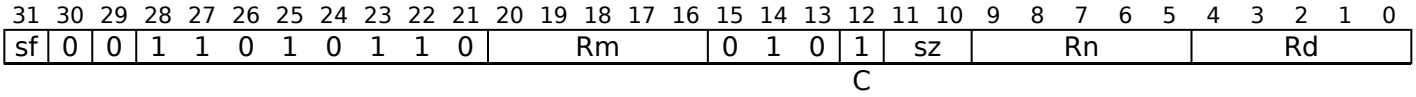
CRC32CB, CRC32CH, CRC32CW, CRC32CX

CRC32 checksum performs a cyclic redundancy check (CRC) calculation on a value held in a general-purpose register. It takes an input CRC value in the first source operand, performs a CRC on the input value in the second source operand, and returns the output CRC value. The second source operand can be 8, 16, 32, or 64 bits. To align with common usage, the bit order of the values is reversed as part of the operation, and the polynomial 0x1EDC6F41 is used for the CRC calculation.
In an Armv8.0 implementation, this is an OPTIONAL instruction. From Armv8.1, it is mandatory for all implementations to implement this instruction.

Note

ID_AA64ISAR0_EL1.CRC32 indicates whether this instruction is supported.

CRC
(FEAT_CRC32)



CRC32CB (sf == 0 && sz == 00)

```
CRC32CB <Wd>, <Wn>, <Wm>
```

CRC32CH (sf == 0 && sz == 01)

```
CRC32CH <Wd>, <Wn>, <Wm>
```

CRC32CW (sf == 0 && sz == 10)

```
CRC32CW <Wd>, <Wn>, <Wm>
```

CRC32CX (sf == 1 && sz == 11)

```
CRC32CX <Wd>, <Wn>, <Xm>
```

```
if !HaveCRCExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sf == '1' && sz != '11' then UNDEFINED;
if sf == '0' && sz == '11' then UNDEFINED;
integer size = 8 << UInt(sz);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose accumulator output register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose accumulator input register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose data source register, encoded in the "Rm" field.
<Wm>	Is the 32-bit name of the general-purpose data source register, encoded in the "Rm" field.

Operation

```
bits(32) acc = X[n, 32];    // accumulator
bits(size) val = X[m, size];    // input value
bits(32) poly = 0x1EDC6F41<31:0>;

bits(32+size) tempacc = BitReverse(acc):Zeros(size);
bits(size+32) tempval = BitReverse(val):Zeros(32);

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
X[d, 32] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
```

Operational information

If PSTATE.DIT is 1:

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CSDB

Consumption of Speculative Data Barrier is a memory barrier that controls speculative execution and data value prediction.

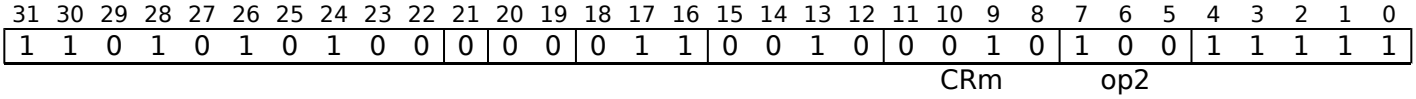
No instruction other than branch instructions appearing in program order after the CSDB can be speculatively executed using the results of any:

- Data value predictions of any instructions.
- PSTATE.{N,Z,C,V} predictions of any instructions other than conditional branch instructions appearing in program order before the CSDB that have not been architecturally resolved.
- Predictions of SVE predication state for any SVE instructions.

Note

For purposes of the definition of CSDB, PSTATE.{N,Z,C,V} is not considered a data value. This definition permits:

- Control flow speculation before and after the CSDB.
- Speculative execution of conditional data processing instructions after the CSDB, unless they use the results of data value or PSTATE.{N,Z,C,V} predictions of instructions appearing in program order before the CSDB that have not been architecturally resolved.



CSDB

```
// Empty.
```

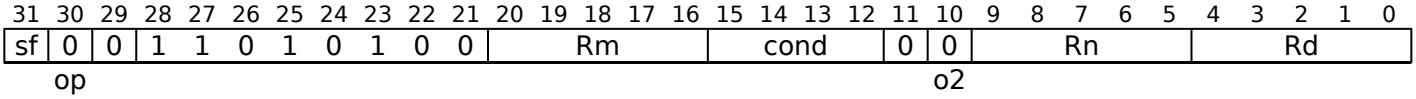
Operation

```
ConsumptionOfSpeculativeDataBarrier();
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CSEL

If the condition is true, Conditional Select writes the value of the first source register to the destination register. If the condition is false, it writes the value of the second source register to the destination register.



32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

Assembler Symbols

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

Operation

```
bits(datasize) result;
if ConditionHolds(cond) then
    result = X[n, datasize];
else
    result = X[m, datasize];

X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

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

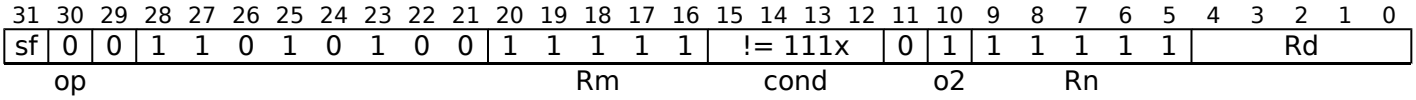
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CSET

Conditional Set sets the destination register to 1 if the condition is TRUE, and otherwise sets it to 0.

This is an alias of CSINC. This means:

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



32-bit (sf == 0)

```
CSET <Wd>, <cond>
```

is equivalent to

```
CSINC <Wd>, WZR, WZR, invert(<cond>)
```

and is always the preferred disassembly.

64-bit (sf == 1)

```
CSET <Xd>, <cond>
```

is equivalent to

```
CSINC <Xd>, XZR, XZR, invert(<cond>)
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<cond>	Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

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

Operational information

If PSTATE.DIT is 1:

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

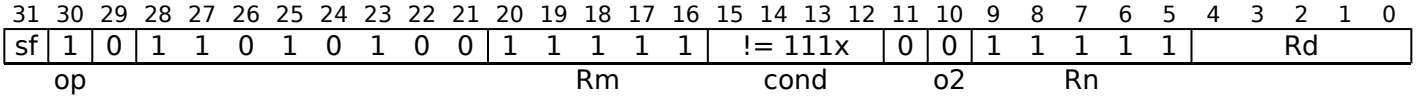
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CSETM

Conditional Set Mask sets all bits of the destination register to 1 if the condition is TRUE, and otherwise sets all bits to 0.

This is an alias of CSINV. This means:

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



32-bit (sf == 0)

```
CSETM <Wd>, <cond>
```

is equivalent to

```
CSINV <Wd>, WZR, WZR, invert(<cond>)
```

and is always the preferred disassembly.

64-bit (sf == 1)

```
CSETM <Xd>, <cond>
```

is equivalent to

```
CSINV <Xd>, XZR, XZR, invert(<cond>)
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<cond>	Is one of the standard conditions, excluding AL and NV, encoded in the "cond" field with its least significant bit inverted.

Operation

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

Operational information

If PSTATE.DIT is 1:

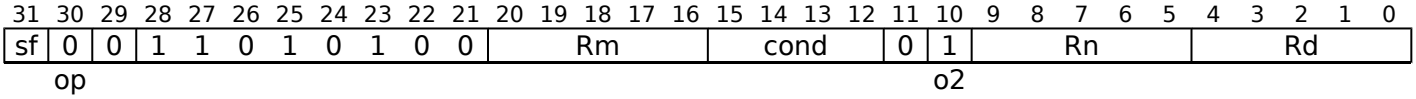
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CSINC

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

This instruction is used by the aliases CINC, and CSET.



32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

Assembler Symbols

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

Alias Conditions

Alias	Is preferred when
CINC	`Rm != '11111' && cond != '111x' && Rn != '11111' && Rn == Rm`
CSET	`Rm == '11111' && cond != '111x' && Rn == '11111'`

Operation

```
bits(datasize) result;
if ConditionHolds(cond) then
    result = X[n, datasize];
else
    result = X[m, datasize];
    result = result + 1;

X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

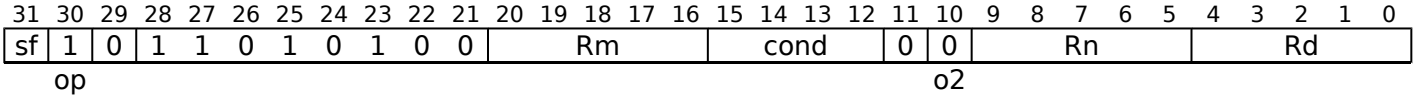
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

- The values of the NZCV flags.

CSINV

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

This instruction is used by the aliases CINV, and CSETM.



32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

Assembler Symbols

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

Alias Conditions

Alias	Is preferred when
CINV	`Rm != '11111' && cond != '111x' && Rn != '11111' && Rn == Rm`
CSETM	`Rm == '11111' && cond != '111x' && Rn == '11111'`

Operation

```
bits(datasize) result;
if ConditionHolds(cond) then
    result = X[n, datasize];
else
    result = X[m, datasize];
    result = NOT(result);

X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

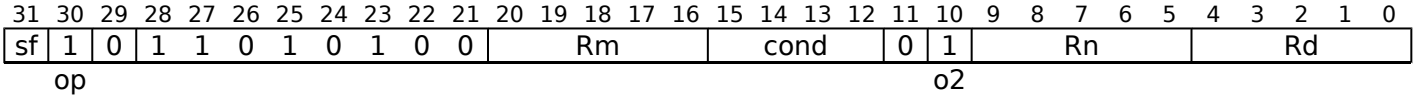
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

- The values of the NZCV flags.

CSNEG

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

This instruction is used by the alias CNEG.



32-bit (sf == 0)

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

64-bit (sf == 1)

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

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

Assembler Symbols

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

Alias Conditions

Alias	Is preferred when
CNEG	`cond != '111x' && Rn == Rm`

Operation

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

X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

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

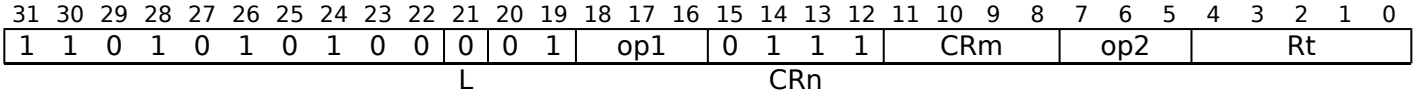
- The values of the NZCV flags.

DC

Data Cache operation. For more information, see *op0==0b01, cache maintenance, TLB maintenance, and address translation instructions*.

This is an alias of SYS. This means:

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



DC <dc_op>, <Xt>

is equivalent to

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

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

Assembler Symbols

<dc_op> Is a DC instruction name, as listed for the DC system instruction group, encoded in “op1:CRm:op2”:

op1	CRm	op2	<dc_op>	Architectural Feature
000	0110	001	IVAC	-
000	0110	010	ISW	-
000	0110	011	IGVAC	FEAT_MTE2
000	0110	100	IGSW	FEAT_MTE2
000	0110	101	IGDVAC	FEAT_MTE2
000	0110	110	IGDSW	FEAT_MTE2
000	1010	010	CSW	-
000	1010	100	CGSW	FEAT_MTE2
000	1010	110	CGDSW	FEAT_MTE2
000	1110	010	CISW	-
000	1110	100	CIGSW	FEAT_MTE2
000	1110	110	CIGDSW	FEAT_MTE2
011	0100	001	ZVA	-
011	0100	011	GVA	FEAT_MTE
011	0100	100	GZVA	FEAT_MTE
011	1010	001	CVAC	-
011	1010	011	CGVAC	FEAT_MTE
011	1010	101	CGDVAC	FEAT_MTE
011	1011	001	CVAU	-
011	1100	001	CVAP	FEAT_DPB
011	1100	011	CGVAP	FEAT_MTE
011	1100	101	CGDVAP	FEAT_MTE
011	1101	001	CVADP	FEAT_DPB2
011	1101	011	CGVADP	FEAT_MTE
011	1101	101	CGDVADP	FEAT_MTE
011	1110	001	CIVAC	-
011	1110	011	CIGVAC	FEAT_MTE
011	1110	101	CIGDVAC	FEAT_MTE

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

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

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

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

Operation

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

DCPS1

Debug Change PE State to EL1, when executed in Debug state:

- If executed at EL0 changes the current Exception level and SP to EL1 using SP_EL1.
- Otherwise, if executed at ELx, selects SP_ELx.

The target exception level of a DCPS1 instruction is:

- EL1 if the instruction is executed at EL0.
- Otherwise, the Exception level at which the instruction is executed.

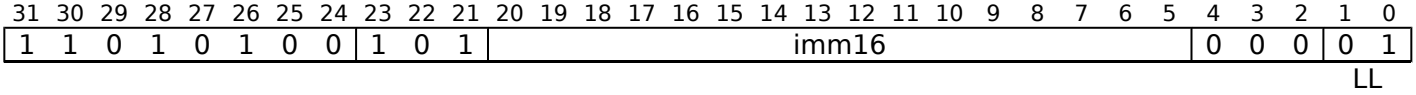
When the target Exception level of a DCPS1 instruction is ELx, on executing this instruction:

- *ELR_ELx* becomes UNKNOWN.
- *SPSR_ELx* becomes UNKNOWN.
- *ESR_ELx* becomes UNKNOWN.
- *DLR_EL0* and *DSPSR_EL0* become UNKNOWN.
- The endianness is set according to *SCTLR_ELx*.EE.

This instruction is UNDEFINED at EL0 in Non-secure state if EL2 is implemented and *HCR_EL2*.TGE == 1.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPS<n> instructions, see *DCPS*.



DCPS1 {#<imm>}

```
if !Halted() then UNDEFINED;
```

Assembler Symbols

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

Operation

```
DCPSInstruction(LL);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

DCPS2

Debug Change PE State to EL2, when executed in Debug state:

- If executed at EL0 or EL1 changes the current Exception level and SP to EL2 using SP_EL2.
- Otherwise, if executed at ELx, selects SP_ELx.

The target exception level of a DCPS2 instruction is:

- EL2 if the instruction is executed at an exception level that is not EL3.
- EL3 if the instruction is executed at EL3.

When the target Exception level of a DCPS2 instruction is ELx, on executing this instruction:

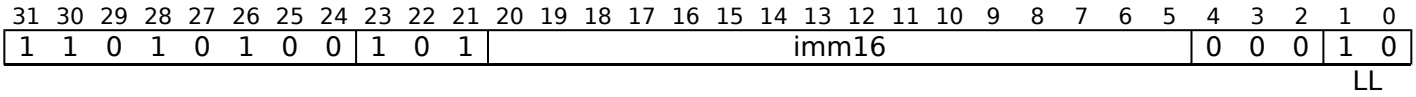
- *ELR_ELx* becomes UNKNOWN.
- *SPSR_ELx* becomes UNKNOWN.
- *ESR_ELx* becomes UNKNOWN.
- *DLR_EL0* and *DSPSR_EL0* become UNKNOWN.
- The endianness is set according to *SCTLR_ELx*.EE.

This instruction is UNDEFINED at the following exception levels:

- All exception levels if EL2 is not implemented.
- At EL0 and EL1 if EL2 is disabled in the current Security state.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPS<n> instructions, see *DCPS*.



DCPS2 {#<imm>}

```
if !Halted() then UNDEFINED;
```

Assembler Symbols

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

Operation

```
DCPSInstruction(LL);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

DCPS3

Debug Change PE State to EL3, when executed in Debug state:

- If executed at EL3 selects SP_EL3.
- Otherwise, changes the current Exception level and SP to EL3 using SP_EL3.

The target exception level of a DCPS3 instruction is EL3.

On executing a DCPS3 instruction:

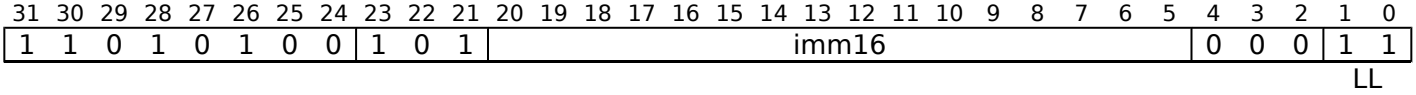
- *ELR_EL3* becomes UNKNOWN.
- *SPSR_EL3* becomes UNKNOWN.
- *ESR_EL3* becomes UNKNOWN.
- *DLR_EL0* and *DSPSR_EL0* become UNKNOWN.
- The endianness is set according to *SCTLR_EL3*.EE.

This instruction is UNDEFINED at all exception levels if either:

- *EDSCR*.SDD == 1.
- EL3 is not implemented.

This instruction is always UNDEFINED in Non-debug state.

For more information on the operation of the DCPS<n> instructions, see *DCPS*.



DCPS3 {#<imm>}

```
if !Halted() then UNDEFINED;
```

Assembler Symbols

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

Operation

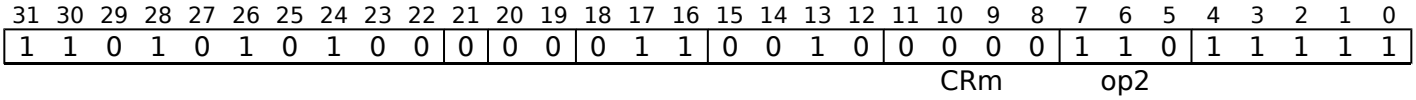
```
DCPSInstruction(LL);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

DGH

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

System
(FEAT_DGH)



DGH

```
if !HaveDGHExt() then EndOfInstruction();
```

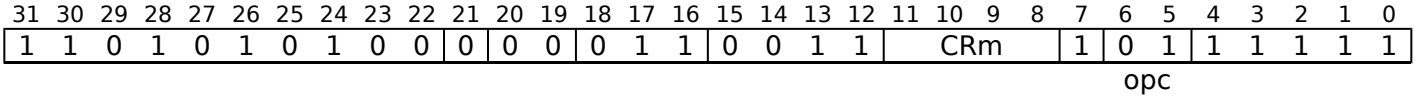
Operation

```
Hint_DGH();
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

DMB

Data Memory Barrier is a memory barrier that ensures the ordering of observations of memory accesses, see *Data Memory Barrier*.



DMB <option>|#<imm>

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

Assembler Symbols

<option> Specifies the limitation on the barrier operation. Values are:

SY
Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm = 0b1111.

ST
Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1110.

LD
Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1101.

ISH
Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b1011.

ISHST
Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1010.

ISHLD
Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1001.

NSH
Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm = 0b0111.

NSHST
Non-shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0110.

NSHLD
Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0101.

OSH
Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b0011.

OSHST
Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0010.

OSHLD
Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0001.

All other encodings of CRm that are not listed above are reserved, and can be encoded using the #<imm> syntax. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior. For more information on whether an access is before or after a barrier instruction, see *Data Memory Barrier (DMB)* or see *Data Synchronization Barrier (DSB)*.

<imm> Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

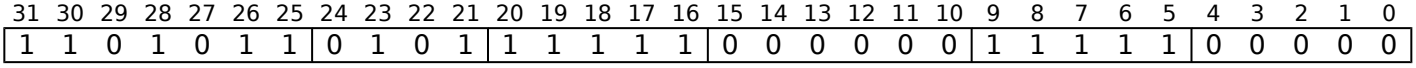
Operation

```
DataMemoryBarrier(domain, types);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

DRPS

Debug restore process state



DRPS

```
if !Halted() || PSTATE.EL == EL0 then UNDEFINED;
```

Operation

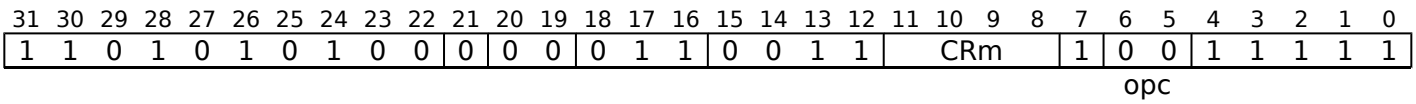
```
DRPSInstruction();
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

DSB

Data Synchronization Barrier is a memory barrier that ensures the completion of memory accesses, see *Data Synchronization Barrier*.
A DSB instruction with the nXS qualifier is complete when the subset of these memory accesses with the XS attribute set to 0 are complete. It does not require that memory accesses with the XS attribute set to 1 are complete.
This instruction is used by the aliases PSSBB, and SSBB.
It has encodings from 2 classes: Memory barrier and Memory nXS barrier

Memory barrier

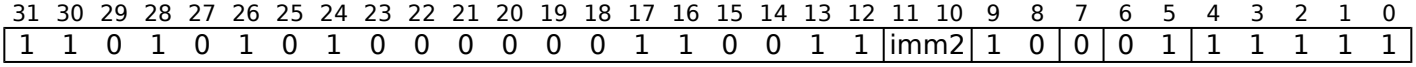


DSB <option>|#<imm>

```
boolean nXS = FALSE;

DSBAlias alias;
case CRm of
    when '0000' alias = DSBAlias_SSBB;
    when '0100' alias = DSBAlias_PSSBB;
    otherwise alias = DSBAlias_DSB;

MBReqDomain domain;
case CRm<3:2> of
    when '00' domain = MBReqDomain_OuterShareable;
    when '01' domain = MBReqDomain_Nonshareable;
    when '10' domain = MBReqDomain_InnerShareable;
    when '11' domain = MBReqDomain_FullSystem;

MBReqTypes types;
case CRm<1:0> of
    when '00' types = MBReqTypes_All; domain = MBReqDomain_FullSystem;
    when '01' types = MBReqTypes_Reads;
    when '10' types = MBReqTypes_Writes;
    when '11' types = MBReqTypes_All;
```

Memory nXS barrier

(FEAT_XS)



DSB <option>nXS

```
if !HaveFeatXS() then UNDEFINED;
MBReqTypes types = MBReqTypes_All;
boolean nXS = TRUE;
DSBAlias alias = DSBAlias_DSB;
MBReqDomain domain;

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

Assembler Symbols

<option> For the memory barrier variant: specifies the limitation on the barrier operation. Values are:

SY
Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as CRm = 0b1111.

ST
Full system is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1110.

LD
Full system is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1101.

ISH
Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b1011.

ISHST
Inner Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b1010.

ISHLD
Inner Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b1001.

NSH
Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as CRm = 0b0111.

NSHST
Non-shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0110.

NSHLD
Non-shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0101.

OSH
Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as CRm = 0b0011.

OSHST
Outer Shareable is the required shareability domain, writes are the required access type, both before and after the barrier instruction. Encoded as CRm = 0b0010.

OSHLD
Outer Shareable is the required shareability domain, reads are the required access type before the barrier instruction, and reads and writes are the required access types after the barrier instruction. Encoded as CRm = 0b0001.

All other encodings of CRm, other than the values 0b0000 and 0b0100, that are not listed above are reserved, and can be encoded using the #<imm> syntax. All unsupported and reserved options must execute as a full system barrier operation, but software must not rely on this behavior. For more information on whether an access is before or after a barrier instruction, see *Data Memory Barrier (DMB)* or see *Data Synchronization Barrier (DSB)*.

Note

The value 0b0000 is used to encode SSBB and the value 0b0100 is used to encode PSSBB.

For the memory nXS barrier variant: specifies the limitation on the barrier operation. Values are:

SY
Full system is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. This option is referred to as the full system barrier. Encoded as imm2 = 0b11.

ISH
Inner Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as imm2 = 0b10.

NSH
Non-shareable is the required shareability domain, reads and writes are the required access, both before and after the barrier instruction. Encoded as imm2 = 0b01.

OSH
Outer Shareable is the required shareability domain, reads and writes are the required access types, both before and after the barrier instruction. Encoded as imm2 = 0b00.

<imm> Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field.

Alias Conditions

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

Operation

```
case alias of
    when DSBAlias_SSBB
        SpeculativeStoreBypassBarrierToVA();
    when DSBAlias_PSSBB
        SpeculativeStoreBypassBarrierToPA();
    when DSBAlias_DSB
        if HaveTME() && TSTATE.depth > 0 then
            FailTransaction(TMFailure_ERR, FALSE);
        if !nXS && HaveFeatXS() && HaveFeatHCX() then
            nXS = PSTATE.EL IN {EL0, EL1} && IsHCRXEL2Enabled() && HCRX_EL2.FnXS == '1';
        DataSynchronizationBarrier(domain, types, nXS);
    otherwise
        Unreachable();
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

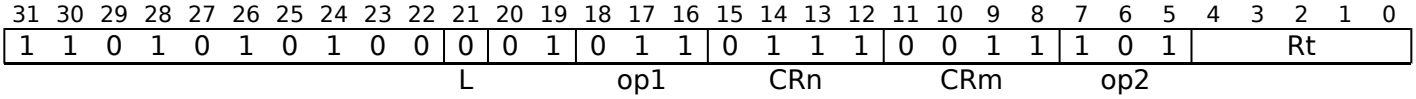
DVP

Data Value Prediction Restriction by Context prevents data value predictions that predict execution addresses based on information gathered from earlier execution within a particular execution context. Data value predictions determined by the actions of code in the target execution context or contexts appearing in program order before the instruction cannot be used to exploitatively control speculative execution occurring after the instruction is complete and synchronized.
For more information, see *DVP RCTX, Data Value Prediction Restriction by Context*.

This is an alias of SYS. This means:

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

System
(FEAT_SPECRES)



DVP RCTX, <Xt>

is equivalent to

SYS #3, C7, C3, #5, <Xt>

and is always the preferred disassembly.

Assembler Symbols

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

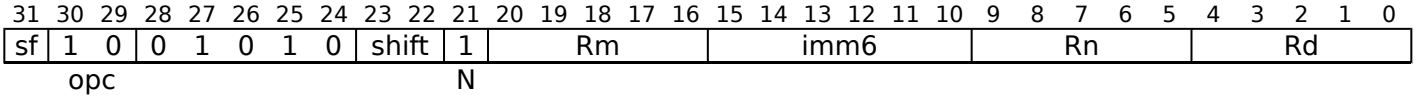
Operation

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

EON (shifted register)

Bitwise Exclusive OR NOT (shifted register) performs a bitwise Exclusive OR NOT of a register value and an optionally-shifted register value, and writes the result to the destination register.



32-bit (sf == 0)

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

64-bit (sf == 1)

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

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

Assembler Symbols

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

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

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

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

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

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

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

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

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

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

Operation

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

operand2 = NOT(operand2);

result = operand1 EOR operand2;

X[d, datasize] = result;
```

Operational information

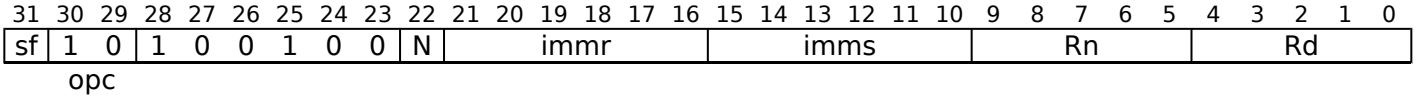
If PSTATE.DIT is 1:

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

EOR (immediate)

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



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

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

64-bit (sf == 1)

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE, datasize);
```

Assembler Symbols

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

Operation

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

result = operand1 EOR imm;

if d == 31 then
    SP[] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

Operational information

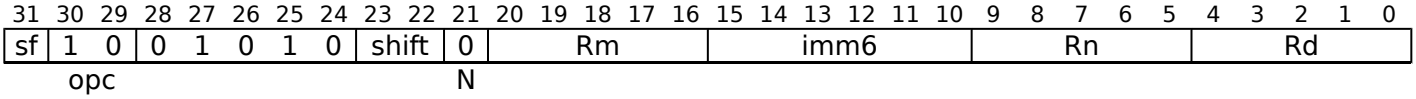
If PSTATE.DIT is 1:

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

EOR (shifted register)

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



32-bit (sf == 0)

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

64-bit (sf == 1)

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

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

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

Assembler Symbols

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

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

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

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

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

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

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

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

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

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

Operation

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

result = operand1 EOR operand2;

X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

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

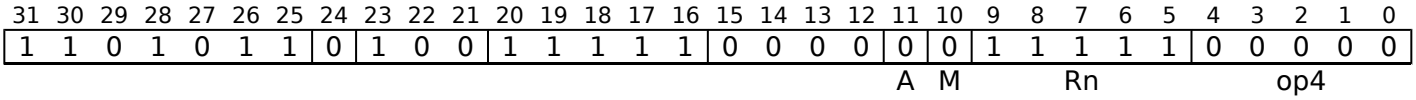
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ERET

Exception Return using the ELR and SPSR for the current Exception level. When executed, the PE restores *PSTATE* from the SPSR, and branches to the address held in the ELR.

The PE checks the SPSR for the current Exception level for an illegal return event. See *Illegal return events from AArch64 state*.

ERET is UNDEFINED at EL0.



ERET

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

Operation

```
AArch64.CheckForERetTrap(FALSE, TRUE);
bits(64) target = ELR[];

AArch64.ExceptionReturn(target, SPSR[]);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ERETAA, ERETAB

Exception Return, with pointer authentication. This instruction authenticates the address in ELR, using SP as the modifier and the specified key, the PE restores *PSTATE* from the SPSR for the current Exception level, and branches to the authenticated address.

Key A is used for ERETAA, and key B is used for ERETAB.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

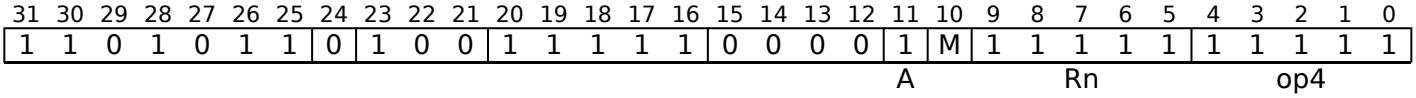
The authenticated address is not written back to ELR.

The PE checks the SPSR for the current Exception level for an illegal return event. See *Illegal return events from AArch64 state*.

ERETAA and ERETAB are UNDEFINED at EL0.

Integer

(FEAT_PAuth)



ERETAA (M == 0)

ERETAA

ERETAB (M == 1)

ERETAB

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

if !HavePACExt() then
    UNDEFINED;
```

Operation

```
AArch64.CheckForERetTrap(TRUE, use_key_a);
bits(64) target = ELR[];
bits(64) modifier = SP[];

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

AArch64.ExceptionReturn(target, SPSR[]);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ESB

Error Synchronization Barrier is an error synchronization event that might also update DISR_EL1 and VDISR_EL2.

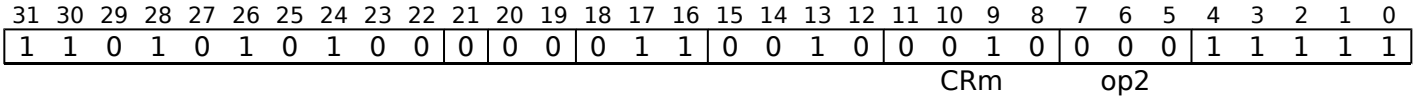
This instruction can be used at all Exception levels and in Debug state.

In Debug state, this instruction behaves as if SError interrupts are masked at all Exception levels. See Error Synchronization Barrier in the Arm(R) Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for Armv8-A architecture profile.

If the RAS Extension is not implemented, this instruction executes as a `NOP`.

System

(FEAT_RAS)



ESB

```
if !HaveRASExt() then EndOfInstruction();
```

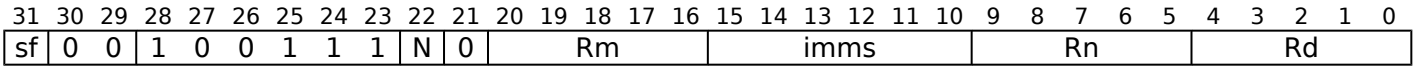
Operation

```
if HaveTME() && TSTATE.depth > 0 then
    FailTransaction(TMFailure_ERR, FALSE);
SynchronizeErrors();
AArch64.ESBOperation();
if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch64.vESBOperation();
TakeUnmaskedSErrorInterrupts();
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

EXTR

Extract register extracts a register from a pair of registers.
This instruction is used by the alias ROR (immediate).



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

```
EXTR <Wd>, <Wn>, <Wm>, #<lsb>
```

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

```
EXTR <Xd>, <Xn>, <Xm>, #<lsb>
```

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

if N != sf then UNDEFINED;
if sf == '0' && imms<5> == '1' then UNDEFINED;
lsb = UInt(imms);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.
<lsb>	For the 32-bit variant: is the least significant bit position from which to extract, in the range 0 to 31, encoded in the "imms" field. For the 64-bit variant: is the least significant bit position from which to extract, in the range 0 to 63, encoded in the "imms" field.

Alias Conditions

Alias	Is preferred when
ROR (immediate)	`Rn == Rm`

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = X[m, datasize];
bits(2*datasize) concat = operand1:operand2;

result = concat<lsb+datasize-1:lsb>;

X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

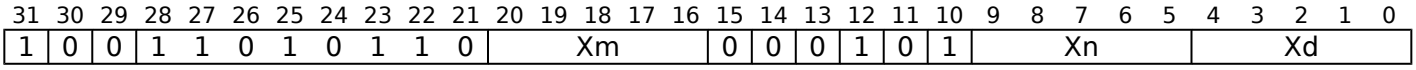
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

GMI

Tag Mask Insert inserts the tag in the first source register into the excluded set specified in the second source register, writing the new excluded set to the destination register.

Integer
(FEAT_MTE)



GMI <Xd>, <Xn|SP>, <Xm>

```
if !HaveMTEExt() then UNDEFINED;
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);
```

Assembler Symbols

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

Operation

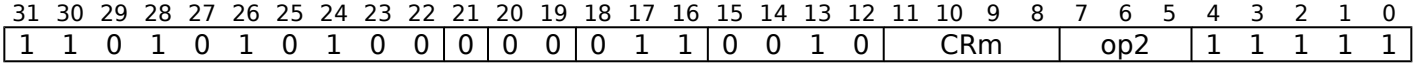
```
bits(64) address = if n == 31 then SP[] else X[n, 64];
bits(64) mask = X[m, 64];
bits(4) tag = AArch64.AllocationTagFromAddress(address);

mask<UInt(tag)> = '1';
X[d, 64] = mask;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

HINT

Hint instruction is for the instruction set space that is reserved for architectural hint instructions.

Some encodings described here are not allocated in this revision of the architecture, and behave as NOPs. These encodings might be allocated to other hint functionality in future revisions of the architecture and therefore must not be used by software.



HINT #<imm>

```
SystemHintOp op;

case CRm:op2 of
    when '0000 000' op = SystemHintOp_NOP;
    when '0000 001' op = SystemHintOp_YIELD;
    when '0000 010' op = SystemHintOp_WFE;
    when '0000 011' op = SystemHintOp_WFI;
    when '0000 100' op = SystemHintOp_SEV;
    when '0000 101' op = SystemHintOp_SEVL;
    when '0000 110'
        if !HaveDGHExt() then EndOfInstruction();    // Instruction executes as NOP
        op = SystemHintOp_DGH;
    when '0000 111' SEE "XPACLRI";
    when '0001 xxx'
        case op2 of
            when '000' SEE "PACIA1716";
            when '010' SEE "PACIB1716";
            when '100' SEE "AUTIA1716";
            when '110' SEE "AUTIB1716";
            otherwise EndOfInstruction();
    when '0010 000'
        if !HaveRASExt() then EndOfInstruction();    // Instruction executes as NOP
        op = SystemHintOp_ESB;
    when '0010 001'
        if !HaveStatisticalProfiling() then EndOfInstruction();    // Instruction executes as NOP
        op = SystemHintOp_PSB;
    when '0010 010'
        if !HaveSelfHostedTrace() then EndOfInstruction();    // Instruction executes as NOP
        op = SystemHintOp_TSB;
    when '0010 100'
        op = SystemHintOp_CSDB;
    when '0011 xxx'
        case op2 of
            when '000' SEE "PACIAZ";
            when '001' SEE "PACIASP";
            when '010' SEE "PACIBZ";
            when '011' SEE "PACIBSP";
            when '100' SEE "AUTIAZ";
            when '101' SEE "AUTIASP";
            when '110' SEE "AUTIBZ";
            when '111' SEE "AUTIBSP";
    when '0100 xx0'
        op = SystemHintOp_BTI;
        // Check branch target compatibility between BTI instruction and PSTATE.BTYPE
        SetBTypeCompatible(BTypeCompatible_BTI(op2<2:1>));
    otherwise EndOfInstruction();
```

Assembler Symbols

<imm>	Is a 7-bit unsigned immediate, in the range 0 to 127 encoded in the "CRm:op2" field. The encodings that are allocated to architectural hint functionality are described in the "Hints" table in the "Index by Encoding".

Note

For allocated encodings of "CRm:op2":

- A disassembler will disassemble the allocated instruction, rather than the HINT instruction.
- An assembler may support assembly of allocated encodings using HINT with the corresponding <imm> value, but it is not required to do so.

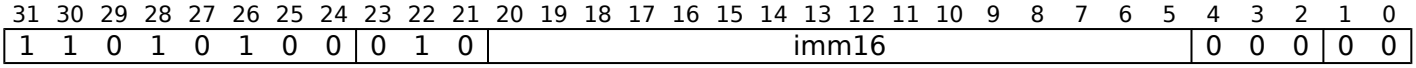
Operation

```
case op of
    when SystemHintOp_YIELD
        Hint_Yield();

    when SystemHintOp_DGH
        Hint_DGH();

    when SystemHintOp_WFE
        integer localtimeout = 1 << 64;    // No local timeout event is generated
        Hint_WFE(localtimeout, WFxType_WFE);

    when SystemHintOp_WFI
        integer localtimeout = 1 << 64;    // No local timeout event is generated
        Hint_WFI(localtimeout, WFxType_WFI);

    when SystemHintOp_SEV
        SendEvent();

    when SystemHintOp_SEVL
        SendEventLocal();

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

    when SystemHintOp_PSB
        ProfilingSynchronizationBarrier();

    when SystemHintOp_TSB
        TraceSynchronizationBarrier();

    when SystemHintOp_CSDB
        ConsumptionOfSpeculativeDataBarrier();

    when SystemHintOp_BTI
        SetBTypeNext('00');

    otherwise    // do nothing
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

HLT

Halt instruction. An HLT instruction can generate a Halt Instruction debug event, which causes entry into Debug state.



HLT #<imm>

```
if EDSCR.HDE == '0' || !HaltingAllowed() then UNDEFINED;
if HaveBTIExt() then
    SetBTypeCompatible(TRUE);
```

Assembler Symbols

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

Operation

```
Halt(DebugHalt_HaltInstruction, FALSE);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

HVC

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

The HVC instruction is UNDEFINED:

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

On executing an HVC instruction, the PE records the exception as a Hypervisor Call exception in *ESR_ELx*, using the EC value 0x16, and the value of the immediate argument.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	0	0	0	0	imm16																0	0	0	1	0

HVC #<imm>

```
// Empty.
```

Assembler Symbols

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

Operation

```
if !HaveEL(EL2) || PSTATE.EL == EL0 || (PSTATE.EL == EL1 && !EL2Enabled()) then
    UNDEFINED;

hvc_enable = if HaveEL(EL3) then SCR_EL3.HCE else NOT(HCR_EL2.HCD);

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

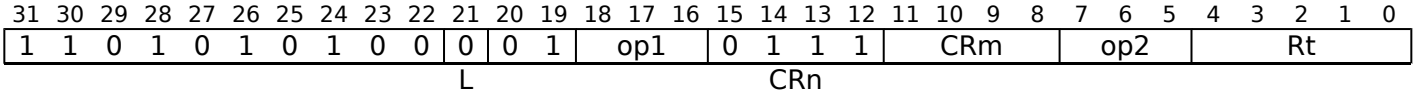
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

IC

Instruction Cache operation. For more information, see *op0==0b01, cache maintenance, TLB maintenance, and address translation instructions*.

This is an alias of SYS. This means:

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



IC <ic_op>{, <Xt>}

is equivalent to

SYS #<op1>, C7, <Cm>, #<op2>{, <Xt>}

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

Assembler Symbols

<ic_op> Is an IC instruction name, as listed for the IC system instruction pages, encoded in “op1:CRm:op2”:

op1	CRm	op2	<ic_op>
000	0001	000	IALLUIS
000	0101	000	IALLU
011	0101	001	IVAU

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

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

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

<Xt> Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

Operation

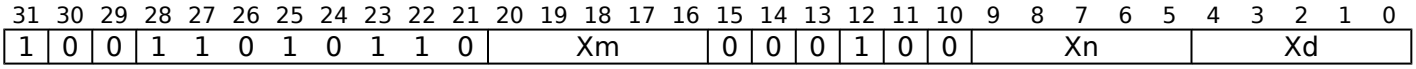
The description of SYS gives the operational pseudocode for this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

IRG

Insert Random Tag inserts a random Logical Address Tag into the address in the first source register, and writes the result to the destination register. Any tags specified in the optional second source register or in GCR_EL1.Exclude are excluded from the selection of the random Logical Address Tag.

Integer
(FEAT_MTE)



IRG <Xd|SP>, <Xn|SP>{, <Xm>}

```
if !HaveMTEExt() then UNDEFINED;
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);
```

Assembler Symbols

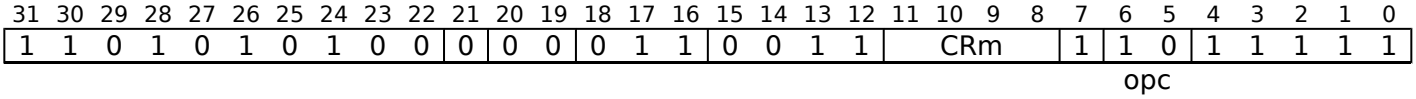
<Xd|SP>	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Xd" field.
<Xn|SP>	Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Xm" field. Defaults to XZR if absent.

Operation

```
bits(64) operand = if n == 31 then SP[] else X[n, 64];
bits(64) exclude_reg = X[m, 64];
bits(16) exclude = exclude_reg<15:0> OR GCR_EL1.Exclude;
bits(4) rtag;

if AArch64.AllocationTagAccessIsEnabled(AccType_NORMAL) then
    if GCR_EL1.RRND == '1' then
        if IsOnes(exclude) then
            rtag = '0000';
        else
            rtag = ChooseRandomNonExcludedTag(exclude);
    else
        bits(4) start = RGSR_EL1.TAG;
        bits(4) offset = AArch64.RandomTag();

        rtag = AArch64.ChooseNonExcludedTag(start, offset, exclude);

        RGSR_EL1.TAG = rtag;
else
    rtag = '0000';

bits(64) result = AArch64.AddressWithAllocationTag(operand, AccType_NORMAL, rtag);

if d == 31 then
    SP[] = result;
else
    X[d, 64] = result;
```

ISB

Instruction Synchronization Barrier flushes the pipeline in the PE and is a context synchronization event. For more information, see *Instruction Synchronization Barrier (ISB)*.




```
ISB {<option>|#<imm>}
```

```
// No additional decoding required
```

Assembler Symbols

<option> Specifies an optional limitation on the barrier operation. Values are:

SY
Full system barrier operation, encoded as CRm = 0b1111. Can be omitted.

All other encodings of CRm are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software.

<imm> Is an optional 4-bit unsigned immediate, in the range 0 to 15, defaulting to 15 and encoded in the "CRm" field.

Operation

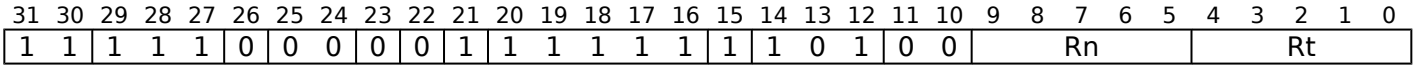
```
InstructionSynchronizationBarrier();
if HaveBRBExt() && BRBEBranchOnISB() then
    BRBEISB();
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD64B

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

Integer
(FEAT_LS64)



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

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

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

Assembler Symbols

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

Operation

```
CheckLDST64BEnabled();

bits(512) data;
bits(64) address;
bits(64) value;
acctype = AccType_ATOMICLS64;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemLoad64B(address, acctype);

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDADD, LDADDA, LDADDAL, LDADDL

Atomic add on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, `LDADDA` and `LDADDAL` load from memory with acquire semantics.
- `LDADDL` and `LDADDAL` store to memory with release semantics.
- `LDADD` has neither acquire nor release semantics.

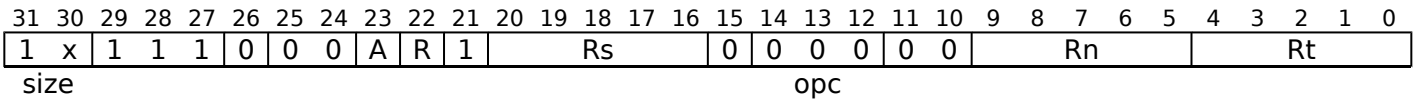
For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STADD, STADDL.

Integer

(FEAT_LSE)



32-bit LDADD (size == 10 && A == 0 && R == 0)

```
LDADD <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDADDA (size == 10 && A == 1 && R == 0)

```
LDADDA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDADDAL (size == 10 && A == 1 && R == 1)

```
LDADDAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDADDL (size == 10 && A == 0 && R == 1)

```
LDADDL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDADD (size == 11 && A == 0 && R == 0)

```
LDADD <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDADDA (size == 11 && A == 1 && R == 0)

```
LDADDA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDADDAL (size == 11 && A == 1 && R == 1)

```
LDADDAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDADDL (size == 11 && A == 0 && R == 1)

```
LDADDL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STADD, STADDL	`A == '0' && Rt == '11111'`

Operation

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

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, datasize];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_ADD, value, ldacctype, stacctype);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDADDB, LDADDAB, LDADDALB, LDADDLB

Atomic add on byte in memory atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDADDAB and LDADDALB load from memory with acquire semantics.
- LDADDLB and LDADDALB store to memory with release semantics.
- LDADDB has neither acquire nor release semantics.

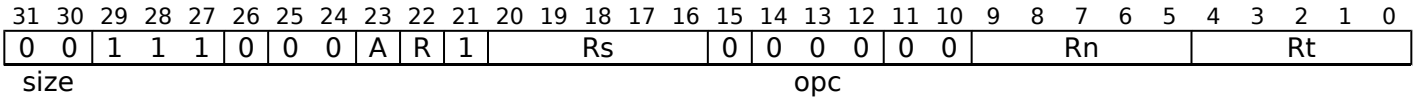
For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STADDB, STADDLB.

Integer

(FEAT_LSE)



LDADDAB (A == 1 && R == 0)

```
LDADDAB <Ws>, <Wt>, [<Xn|SP>]
```

LDADDALB (A == 1 && R == 1)

```
LDADDALB <Ws>, <Wt>, [<Xn|SP>]
```

LDADDB (A == 0 && R == 0)

```
LDADDB <Ws>, <Wt>, [<Xn|SP>]
```

LDADDLB (A == 0 && R == 1)

```
LDADDLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

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

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STADDB, STADDLB	`A == '0' && Rt == '11111'`

Operation

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

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, 8];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_ADD, value, ldacctype, stacctype);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDADDH, LDADDAH, LDADDALH, LDADDLH

Atomic add on halfword in memory atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDADDAH and LDADDALH load from memory with acquire semantics.
- LDADDLH and LDADDALH store to memory with release semantics.
- LDADDH has neither acquire nor release semantics.

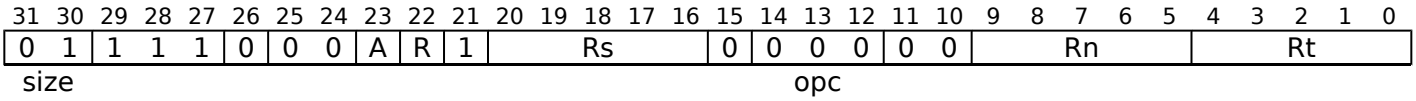
For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STADDH, STADDLH.

Integer

(FEAT_LSE)



LDADDAH (A == 1 && R == 0)

```
LDADDAH <Ws>, <Wt>, [<Xn|SP>]
```

LDADDALH (A == 1 && R == 1)

```
LDADDALH <Ws>, <Wt>, [<Xn|SP>]
```

LDADDH (A == 0 && R == 0)

```
LDADDH <Ws>, <Wt>, [<Xn|SP>]
```

LDADDLH (A == 1 && R == 1)

```
LDADDLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

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

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STADDH, STADDLH	`A == '0' && Rt == '11111'`

Operation

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

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, 16];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_ADD, value, ldacctype, stacctype);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDAPR

Load-Acquire RCpc Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from the derived address in memory, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

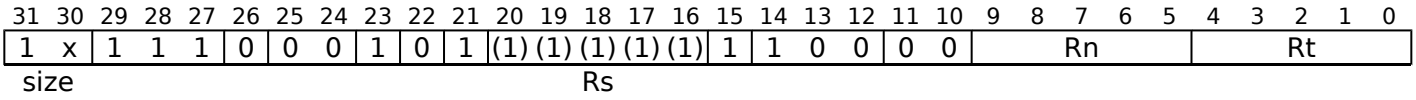
- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

Integer

(FEAT_LRCPC)



32-bit (size == 10)

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

64-bit (size == 11)

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

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

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

Assembler Symbols

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

Operation

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

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = Mem[address, dbytes, AccType_ORDERED];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDAPRB

Load-Acquire RCpc Register Byte derives an address from a base register value, loads a byte from the derived address in memory, zero-extends it and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

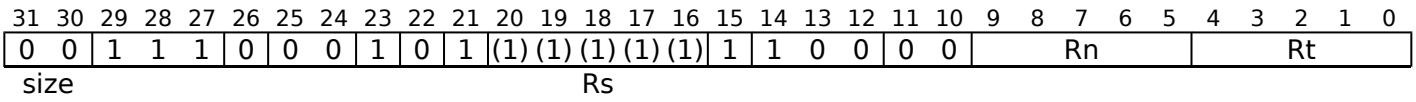
- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

Integer

(FEAT_LRCPC)




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

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

boolean tag_checked = n != 31;
```

Assembler Symbols

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

Operation

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

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = Mem[address, 1, AccType_ORDERED];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDAPRH

Load-Acquire RCpc Register Halfword derives an address from a base register value, loads a halfword from the derived address in memory, zero-extends it and writes it to a register.

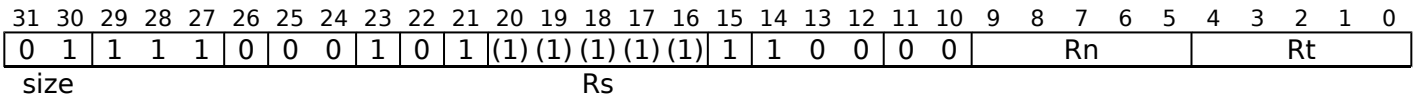
The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

Integer
(FEAT_LRCPC)




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

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

boolean tag_checked = n != 31;
```

Assembler Symbols

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

Operation

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

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = Mem[address, 2, AccType_ORDERED];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDAPUR

Load-Acquire RCpc Register (unscaled) calculates an address from a base register and an immediate offset, loads a 32-bit word or 64-bit doubleword from memory, zero-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

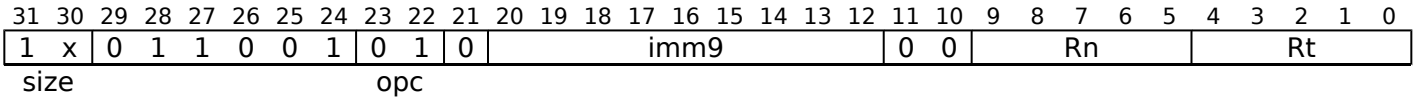
- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

Unscaled offset

(FEAT_LRCPC2)



32-bit (size == 10)

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

64-bit (size == 11)

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

```
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

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

Shared Decode

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

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

Operation

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

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, datasize DIV 8, AccType_ORDERED];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDAPURB

Load-Acquire RCpc Register Byte (unscaled) calculates an address from a base register and an immediate offset, loads a byte from memory, zero-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

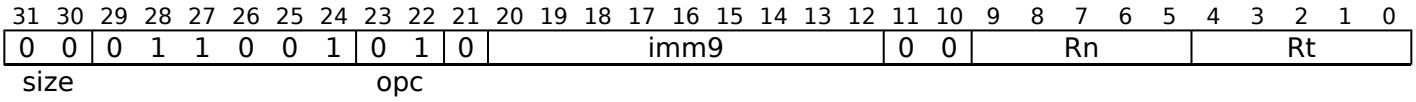
- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

Unscaled offset

(FEAT_LRCPC2)



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

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

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

Shared Decode

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

boolean tag_checked = n != 31;
```

Operation

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

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, 1, AccType_ORDERED];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

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

LDAPURH

Load-Acquire RCpc Register Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a halfword from memory, zero-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

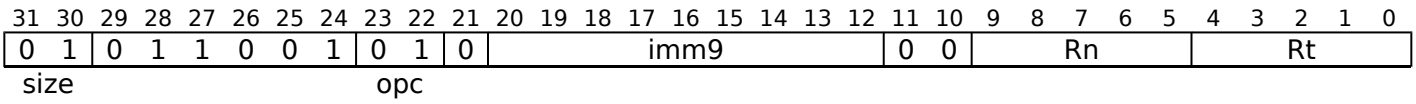
- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

Unscaled offset

(FEAT_LRCPC2)



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

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

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

Shared Decode

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

boolean tag_checked = n != 31;
```

Operation

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

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, 2, AccType_ORDERED];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

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

LDAPURSB

Load-Acquire RCpc Register Signed Byte (unscaled) calculates an address from a base register and an immediate offset, loads a signed byte from memory, sign-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

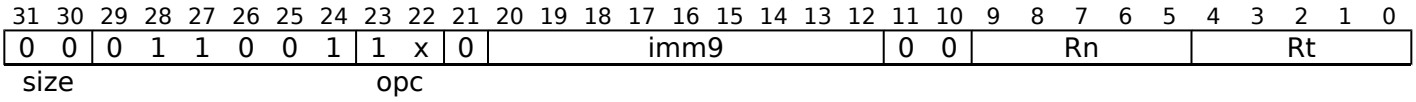
- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

Unscaled offset

(FEAT_LRCPC2)



32-bit (opc == 11)

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

64-bit (opc == 10)

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

bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

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

Shared Decode

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

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

boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);
```

Operation

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

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t, 8];
        Mem[address, 1, AccType_ORDERED] = data;

    when MemOp_LOAD
        data = Mem[address, 1, AccType_ORDERED];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);

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

Operational information

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDAPURSH

Load-Acquire RCpc Register Signed Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a signed halfword from memory, sign-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

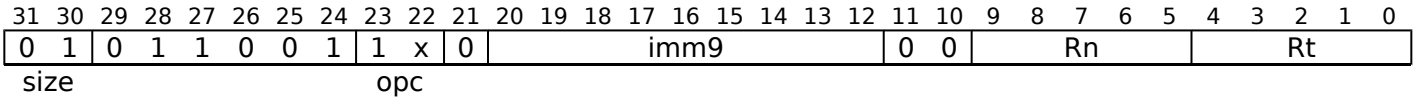
- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

Unscaled offset

(FEAT_LRCPC2)



32-bit (opc == 11)

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

64-bit (opc == 10)

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

bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

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

Shared Decode

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

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

boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);
```

Operation

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

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t, 16];
        Mem[address, 2, AccType_ORDERED] = data;

    when MemOp_LOAD
        data = Mem[address, 2, AccType_ORDERED];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);

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

Operational information

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDAPURSW

Load-Acquire RCpc Register Signed Word (unscaled) calculates an address from a base register and an immediate offset, loads a signed word from memory, sign-extends it, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

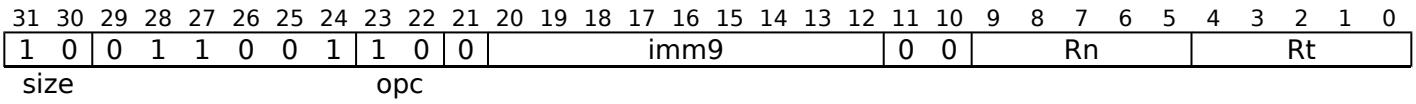
- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

Unscaled offset

(FEAT_LRCPC2)




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

bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

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

Shared Decode

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

boolean tag_checked = n != 31;
```

Operation

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

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, 4, AccType_ORDERED];
X[t, 64] = SignExtend(data, 64);
```

Operational information

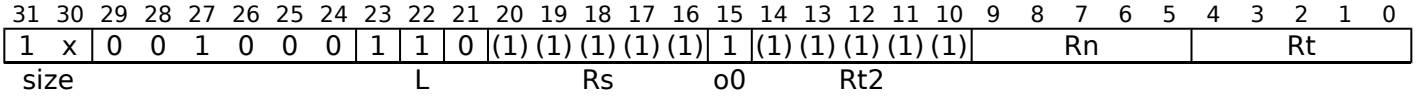
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

LDAR

Load-Acquire Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

Note

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.



32-bit (size == 10)

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

64-bit (size == 11)

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

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

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

Assembler Symbols

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

Operation

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

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = Mem[address, dbytes, AccType_ORDERED];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

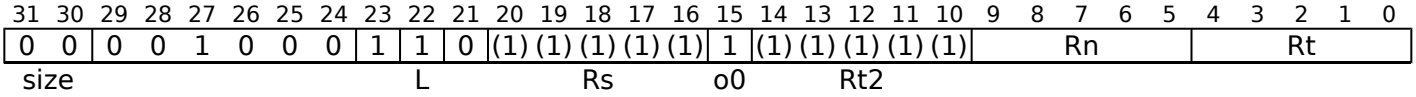
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

LDARB

Load-Acquire Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

Note

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.



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

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

boolean tag_checked = n != 31;
```

Assembler Symbols

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

Operation

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

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = Mem[address, 1, AccType_ORDERED];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

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

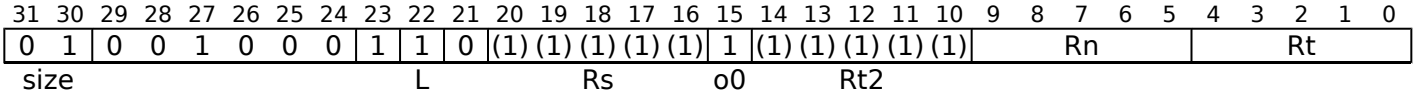
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDARH

Load-Acquire Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it, and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

Note

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.



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

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

boolean tag_checked = n != 31;
```

Assembler Symbols

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

Operation

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

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = Mem[address, 2, AccType_ORDERED];
X[t, 32] = ZeroExtend(data, 32);
```

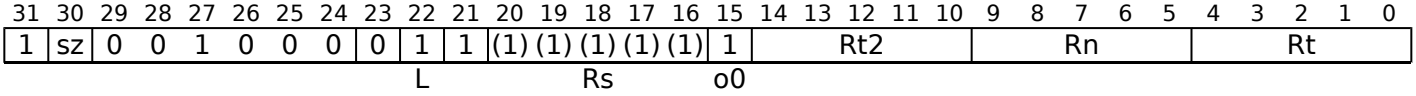
Operational information

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDAXP

Load-Acquire Exclusive Pair of Registers derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information on single-copy atomicity and alignment requirements, see *Requirements for single-copy atomicity* and *Alignment of data accesses*. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. The instruction also has memory ordering semantics, as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.



32-bit (sz == 0)

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

64-bit (sz == 1)

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

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDAXP*.

Assembler Symbols

<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

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

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

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

if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    X[t, datasize] = bits(datasize) UNKNOWN;    // In this case t = t2
elsif elsize == 32 then
    // 32-bit load exclusive pair (atomic)
    data = Mem[address, dbytes, AccType_ORDEREDATOMIC];
    if BigEndian(AccType_ORDEREDATOMIC) then
        X[t, datasize-elsize] = data<datasize-1:elsize>;
        X[t2, elsize] = data<elsize-1:0>;
    else
        X[t, elsize] = data<elsize-1:0>;
        X[t2, datasize-elsize] = data<datasize-1:elsize>;
else // elsize == 64
    // 64-bit load exclusive pair (not atomic),
    // but must be 128-bit aligned
    if address != Align(address, dbytes) then
        AArch64.Abort(address, AlignmentFault(AccType_ORDEREDATOMIC, FALSE, FALSE));
    X[t, 64] = Mem[address, 8, AccType_ORDEREDATOMIC];
    X[t2, 64] = Mem[address+8, 8, AccType_ORDEREDATOMIC];
```

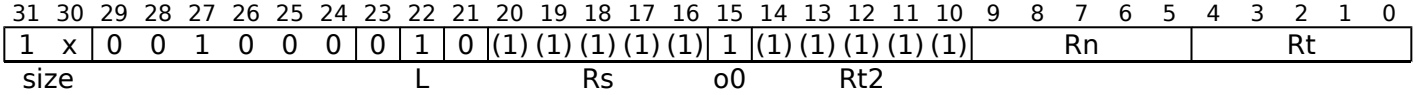
Operational information

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

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDAXR

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



32-bit (size == 10)

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

64-bit (size == 11)

```
LDAXR <Xt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

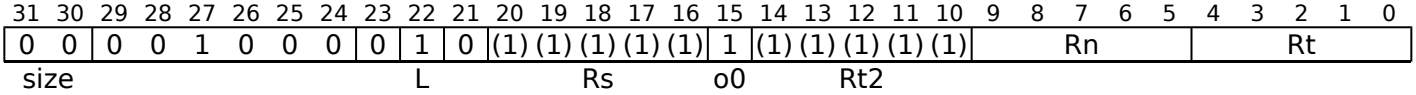
```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

data = Mem[address, dbytes, AccType_ORDEREDATOMIC];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

LDAXRB

Load-Acquire Exclusive Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.



LDAXRB <Wt>, [<Xn|SP>{,#0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 1);

data = Mem[address, 1, AccType_ORDEREDATOMIC];
X[t, 32] = ZeroExtend(data, 32);
```

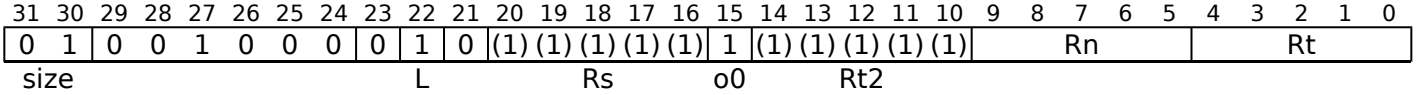
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDAXRH

Load-Acquire Exclusive Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.



LDAXRH <Wt>, [<Xn|SP>{,#0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 2);

data = Mem[address, 2, AccType_ORDEREDATOMIC];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDCLR, LDCLRA, LDCLRAL, LDCLRL

Atomic bit clear on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, `LDCLRA` and `LDCLRAL` load from memory with acquire semantics.
- `LDCLRL` and `LDCLRAL` store to memory with release semantics.
- `LDCLR` has neither acquire nor release semantics.

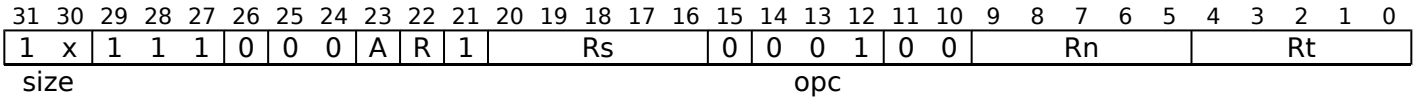
For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STCLR, STCLRL.

Integer

(FEAT_LSE)



32-bit LDCLR (size == 10 && A == 0 && R == 0)

```
LDCLR <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDCLRA (size == 10 && A == 1 && R == 0)

```
LDCLRA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDCLRAL (size == 10 && A == 1 && R == 1)

```
LDCLRAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDCLRL (size == 10 && A == 0 && R == 1)

```
LDCLRL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDCLR (size == 11 && A == 0 && R == 0)

```
LDCLR <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDCLRA (size == 11 && A == 1 && R == 0)

```
LDCLRA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDCLRAL (size == 11 && A == 1 && R == 1)

```
LDCLRAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDCLRL (size == 11 && A == 0 && R == 1)

```
LDCLRL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STCLR, STCLRL	`A == '0' && Rt == '11111'`

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, datasize];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_BIC, value, ldacctype, stacctype);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB

Atomic bit clear on byte in memory atomically loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDCLRAB and LDCLRALB load from memory with acquire semantics.
- LDCLRLB and LDCLRALB store to memory with release semantics.
- LDCLRB has neither acquire nor release semantics.

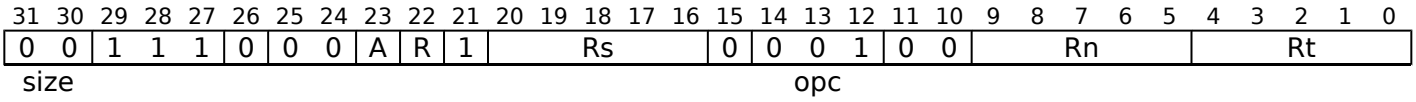
For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STCLRB, STCLRLB.

Integer

(FEAT_LSE)



LDCLRAB (A == 1 && R == 0)

```
LDCLRAB <Ws>, <Wt>, [<Xn|SP>]
```

LDCLRALB (A == 1 && R == 1)

```
LDCLRALB <Ws>, <Wt>, [<Xn|SP>]
```

LDCLRB (A == 0 && R == 0)

```
LDCLRB <Ws>, <Wt>, [<Xn|SP>]
```

LDCLRLB (A == 0 && R == 1)

```
LDCLRLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STCLRB, STCLRLB	`A == '0' && Rt == '11111'`

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, 8];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_BIC, value, ldacctype, stacctype);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH

Atomic bit clear on halfword in memory atomically loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDCLRAH and LDCLRALH load from memory with acquire semantics.
- LDCLRLH and LDCLRALH store to memory with release semantics.
- LDCLRH has neither acquire nor release semantics.

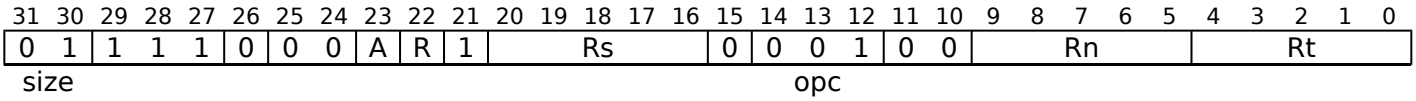
For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STCLRH, STCLRLH.

Integer

(FEAT_LSE)



LDCLRAH (A == 1 && R == 0)

```
LDCLRAH <Ws>, <Wt>, [<Xn|SP>]
```

LDCLRALH (A == 1 && R == 1)

```
LDCLRALH <Ws>, <Wt>, [<Xn|SP>]
```

LDCLRH (A == 0 && R == 0)

```
LDCLRH <Ws>, <Wt>, [<Xn|SP>]
```

LDCLRLH (A == 0 && R == 1)

```
LDCLRLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STCLRH, STCLRLH	`A == '0' && Rt == '11111'`

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, 16];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_BIC, value, ldacctype, stacctype);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDEOR, LDEORA, LDEORAL, LDEORL

Atomic exclusive OR on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, `LDEORA` and `LDEORAL` load from memory with acquire semantics.
- `LDEORL` and `LDEORAL` store to memory with release semantics.
- `LDEOR` has neither acquire nor release semantics.

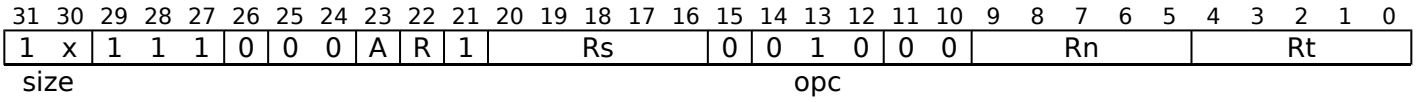
For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STEOR, STEORL.

Integer

(FEAT_LSE)



32-bit LDEOR (size == 10 && A == 0 && R == 0)

```
LDEOR <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDEORA (size == 10 && A == 1 && R == 0)

```
LDEORA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDEORAL (size == 10 && A == 1 && R == 1)

```
LDEORAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDEORL (size == 10 && A == 0 && R == 1)

```
LDEORL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDEOR (size == 11 && A == 0 && R == 0)

```
LDEOR <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDEORA (size == 11 && A == 1 && R == 0)

```
LDEORA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDEORAL (size == 11 && A == 1 && R == 1)

```
LDEORAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDEORL (size == 11 && A == 0 && R == 1)

```
LDEORL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STEOR, STEORL	`A == '0' && Rt == '11111'`

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, datasize];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_EOR, value, ldacctype, stacctype);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDEORB, LDEORAB, LDEORALB, LDEORLB

Atomic exclusive OR on byte in memory atomically loads an 8-bit byte from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDEORAB and LDEORALB load from memory with acquire semantics.
- LDEORLB and LDEORALB store to memory with release semantics.
- LDEORB has neither acquire nor release semantics.

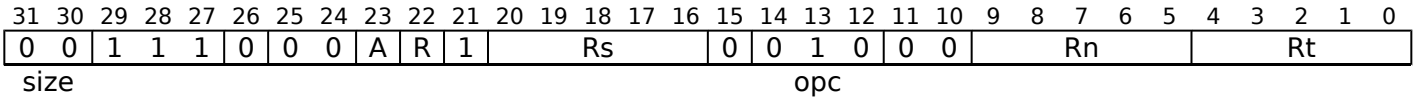
For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STEORB, STEORLB.

Integer

(FEAT_LSE)



LDEORAB (A == 1 && R == 0)

```
LDEORAB <Ws>, <Wt>, [<Xn|SP>]
```

LDEORALB (A == 1 && R == 1)

```
LDEORALB <Ws>, <Wt>, [<Xn|SP>]
```

LDEORB (A == 0 && R == 0)

```
LDEORB <Ws>, <Wt>, [<Xn|SP>]
```

LDEORLB (A == 0 && R == 1)

```
LDEORLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STEORB, STEORLB	`A == '0' && Rt == '11111'`

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, 8];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_EOR, value, ldacctype, stacctype);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDEORH, LDEORAH, LDEORALH, LDEORLH

Atomic exclusive OR on halfword in memory atomically loads a 16-bit halfword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDEORAH and LDEORALH load from memory with acquire semantics.
- LDEORLH and LDEORALH store to memory with release semantics.
- LDEORH has neither acquire nor release semantics.

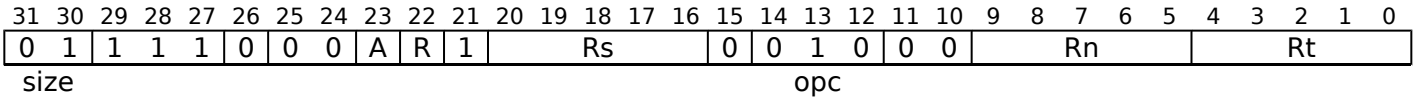
For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STEORH, STEORLH.

Integer

(FEAT_LSE)



LDEORAH (A == 1 && R == 0)

```
LDEORAH <Ws>, <Wt>, [<Xn|SP>]
```

LDEORALH (A == 1 && R == 1)

```
LDEORALH <Ws>, <Wt>, [<Xn|SP>]
```

LDEORH (A == 0 && R == 0)

```
LDEORH <Ws>, <Wt>, [<Xn|SP>]
```

LDEORLH (A == 0 && R == 1)

```
LDEORLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STEORH, STEORLH	`A == '0' && Rt == '11111'`

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, 16];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_EOR, value, ldacctype, stacctype);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

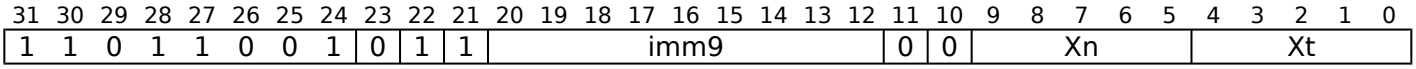
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDG

Load Allocation Tag loads an Allocation Tag from a memory address, generates a Logical Address Tag from the Allocation Tag and merges it into the destination register. The address used for the load is calculated from the base register and an immediate signed offset scaled by the Tag granule.

Integer
(FEAT_MTE)



LDG <Xt>, [<Xn|SP>{, #<simm>}]

```
if !HaveMTEExt() then UNDEFINED;
integer t = UInt(Xt);
integer n = UInt(Xn);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
```

Assembler Symbols

<Xt>	Is the 64-bit name of the general-purpose destination register, encoded in the "Xt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
<simm>	Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

Operation

```
bits(64) address;
bits(4) tag;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;
address = Align(address, TAG_GRANULE);

tag = AArch64.MemTag[address, AccType_NORMAL];
X[t, 64] = AArch64.AddressWithAllocationTag(X[t, 64], AccType_NORMAL, tag);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

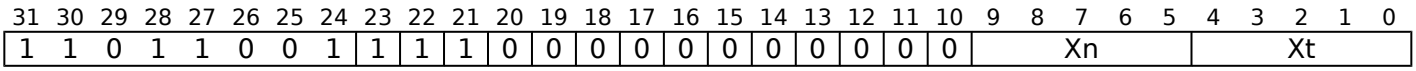
LDGM

Load Tag Multiple reads a naturally aligned block of N Allocation Tags, where the size of N is identified in GMID_EL1.BS, and writes the Allocation Tag read from address A to the destination register at 4*A<7:4>+3:4*A<7:4>. Bits of the destination register not written with an Allocation Tag are set to 0.

This instruction is UNDEFINED at EL0.

This instruction generates an Unchecked access.

Integer
(FEAT_MTE2)



LDGM <Xt>, [<Xn|SP>]

```
if !HaveMTE2Ext() then UNDEFINED;
integer t = UInt(Xt);
integer n = UInt(Xn);
```

Assembler Symbols

<Xt>	Is the 64-bit name of the general-purpose destination register, encoded in the "Xt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

Operation

```
if PSTATE.EL == EL0 then
    UNDEFINED;

bits(64) data = Zeros(64);
bits(64) address;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

integer size = 4 * (2 ^ (UInt(GMID_EL1.BS)));
address = Align(address, size);
integer count = size >> LOG2_TAG_GRANULE;
integer index = UInt(address<LOG2_TAG_GRANULE+3:LOG2_TAG_GRANULE>);

for i = 0 to count-1
    bits(4) tag = AArch64.MemTag[address, AccType_NORMAL];
    data<(index*4)+3:index*4> = tag;
    address = address + TAG_GRANULE;
    index = index + 1;

X[t, 64] = data;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDLAR

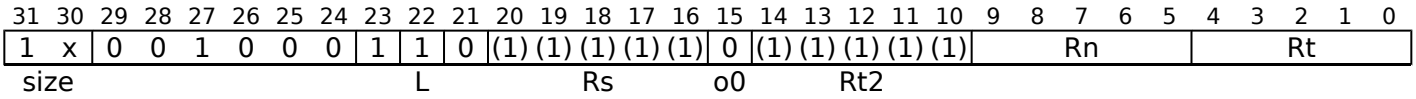
Load LOAcquire Register loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

Note

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

No offset

(FEAT_LOR)



32-bit (size == 10)

```
LDLAR <Wt>, [<Xn|SP>{,#0}]
```

64-bit (size == 11)

```
LDLAR <Xt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = Mem[address, dbytes, AccType_LIMITEDORDERED];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

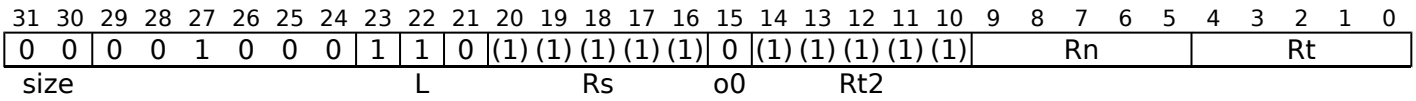
LDLARB

Load LOAcquire Register Byte loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

Note

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

No offset
(FEAT_LOR)



LDLARB <Wt>, [<Xn|SP>{,#0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = Mem[address, 1, AccType_LIMITEDORDERED];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDLARH

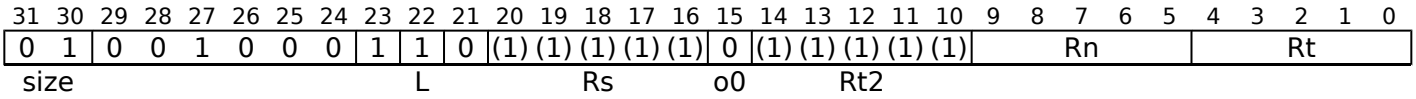
Load LOAcquire Register Halfword loads a halfword from memory, zero-extends it, and writes it to a register. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

Note

For this instruction, if the destination is WZR/XZR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

No offset

(FEAT_LOR)



LDLARH <Wt>, [<Xn|SP>{,#0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = Mem[address, 2, AccType_LIMITEDORDERED];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

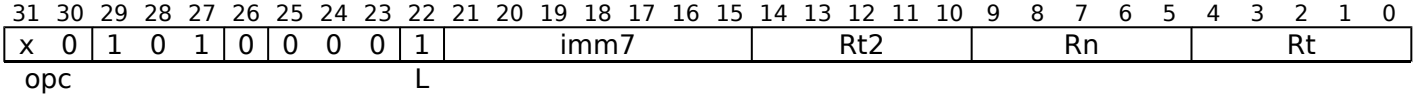
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDNP

Load Pair of Registers, with non-temporal hint, calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers.

For information about memory accesses, see *Load/Store addressing modes*. For information about Non-temporal pair instructions, see *Load/Store Non-temporal pair*.



32-bit (opc == 00)

```
LDNP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

64-bit (opc == 10)

```
LDNP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

```
// Empty.
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDNP*.

Assembler Symbols

<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4. For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc<0> == '1' then UNDEFINED;
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

if HaveLSE2Ext() then
    bits(2*datasize) full_data;
    full_data = Mem[address, 2*dbytes, AccType_NORMAL, TRUE];
    if BigEndian(AccType_STREAM) then
        data2 = full_data<(datasize-1):0>;
        data1 = full_data<(2*datasize-1):datasize>;
    else
        data1 = full_data<(datasize-1):0>;
        data2 = full_data<(2*datasize-1):datasize>;
else
    data1 = Mem[address, dbytes, AccType_STREAM];
    data2 = Mem[address+dbytes, dbytes, AccType_STREAM];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
X[t, datasize] = data1;
X[t2, datasize] = data2;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

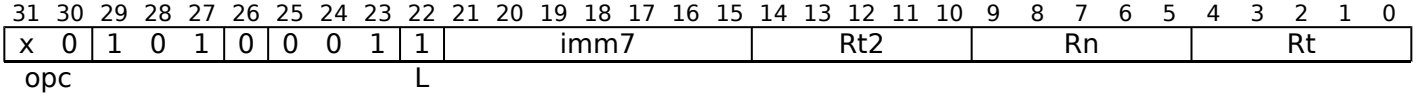
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDP

Load Pair of Registers calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Signed offset

Post-index



32-bit (opc == 00)

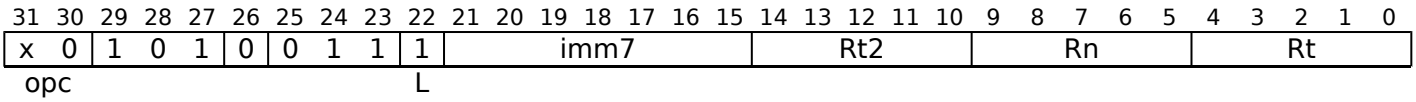
```
LDP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>
```

64-bit (opc == 10)

```
LDP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

Pre-index



32-bit (opc == 00)

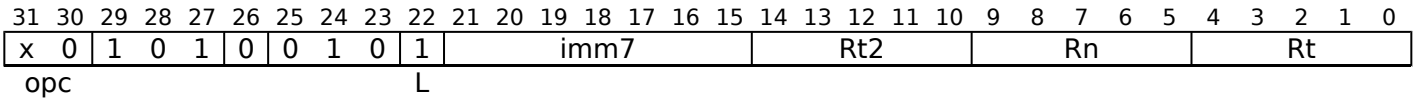
```
LDP <Wt1>, <Wt2>, [<Xn|SP>, #<imm>]!
```

64-bit (opc == 10)

```
LDP <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

Signed offset



32-bit (opc == 00)

```
LDP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

64-bit (opc == 10)

```
LDP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDP*.

Assembler Symbols

<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4. For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4. For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8. For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if L:opc<0> == '01' || opc == '11' then UNDEFINED;
boolean signed = (opc<0> != '0');
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

if HaveLSE2Ext() && !signed then
    bits(2*datasize) full_data;
    full_data = Mem[address, 2*dbytes, AccType_NORMAL, TRUE];
    if BigEndian(AccType_NORMAL) then
        data2 = full_data<(datasize-1):0>;
        data1 = full_data<(2*datasize-1):datasize>;
    else
        data1 = full_data<(datasize-1):0>;
        data2 = full_data<(2*datasize-1):datasize>;
else
    data1 = Mem[address, dbytes, AccType_NORMAL];
    data2 = Mem[address+dbytes, dbytes, AccType_NORMAL];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
if signed then
    X[t, 64] = SignExtend(data1, 64);
    X[t2, 64] = SignExtend(data2, 64);
else
    X[t, datasize] = data1;
    X[t2, datasize] = data2;

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

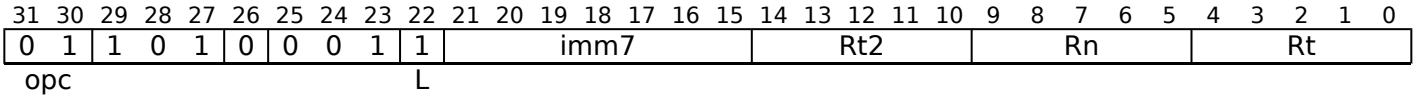
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDPSW

Load Pair of Registers Signed Word calculates an address from a base register value and an immediate offset, loads two 32-bit words from memory, sign-extends them, and writes them to two registers. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Signed offset

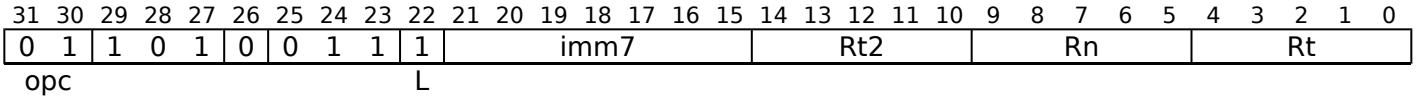
Post-index




```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

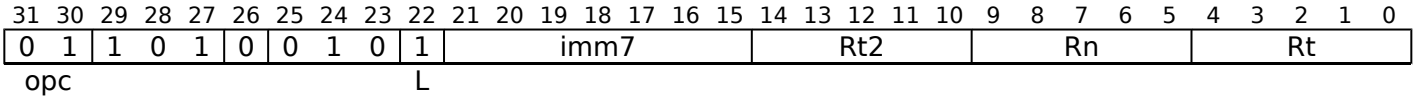
Pre-index




```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

Signed offset




```
LDPSW <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDPSW*.

Assembler Symbols

<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the post-index and pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4. For the signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
bits(64) offset = LSL(SignExtend(imm7, 64), 2);
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;
boolean wb_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Operation

```
bits(64) address;
bits(32) data1;
bits(32) data2;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

data1 = Mem[address, 4, AccType_NORMAL];
data2 = Mem[address+4, 4, AccType_NORMAL];
if rt_unknown then
    data1 = bits(32) UNKNOWN;
    data2 = bits(32) UNKNOWN;
X[t, 64] = SignExtend(data1, 64);
X[t2, 64] = SignExtend(data2, 64);
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

Operational information

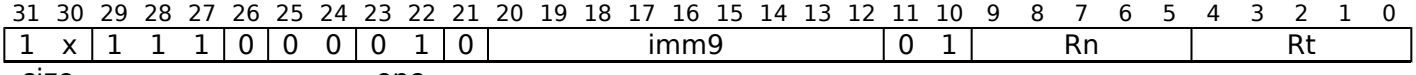
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

LDR (immediate)

Load Register (immediate) loads a word or doubleword from memory and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*. The Unsigned offset variant scales the immediate offset value by the size of the value accessed before adding it to the base register value.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index



32-bit (size == 10)

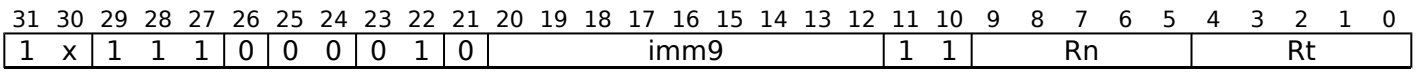
```
LDR <Wt>, [<Xn|SP>], #<simm>
```

64-bit (size == 11)

```
LDR <Xt>, [<Xn|SP>], #<simm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index



32-bit (size == 10)

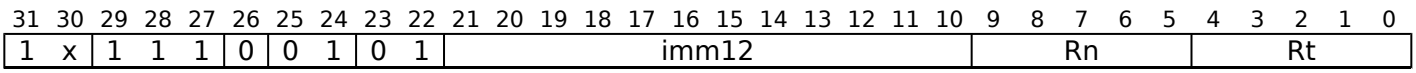
```
LDR <Wt>, [<Xn|SP>, #<simm>]!
```

64-bit (size == 11)

```
LDR <Xt>, [<Xn|SP>, #<simm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



32-bit (size == 10)

```
LDR <Wt>, [<Xn|SP>{, #<pimm>}]
```

64-bit (size == 11)

```
LDR <Xt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDR (immediate)*.

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4. For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tag_checked = wback || n != 31;

boolean wb_unknown = FALSE;
Constraint c;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();
```

Operation

```
bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

data = Mem[address, datasize DIV 8, AccType_NORMAL];
X[t, regsize] = ZeroExtend(data, regsize);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

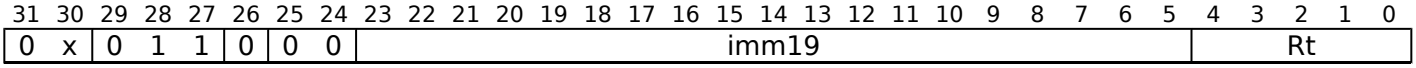
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDR (literal)

Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.



32-bit (opc == 00)

```
LDR <Wt>, <label>
```

64-bit (opc == 01)

```
LDR <Xt>, <label>
```

```
integer t = UInt(Rt);
MemOp memop = MemOp_LOAD;
boolean signed = FALSE;
integer size;
bits(64) offset;

case opc of
    when '00'
        size = 4;
    when '01'
        size = 8;
    when '10'
        size = 4;
        signed = TRUE;
    when '11'
        memop = MemOp_PREFETCH;

offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<label>	Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```
bits(64) address = PC[] + offset;
bits(size*8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);

case memop of
    when MemOp_LOAD
        data = Mem[address, size, AccType_NORMAL];
        if signed then
            X[t, 64] = SignExtend(data, 64);
        else
            X[t, size*8] = data;

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

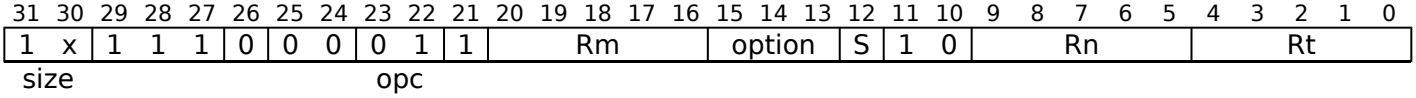
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDR (register)

Load Register (register) calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value can optionally be shifted and extended. For information about memory accesses, see *Load/Store addressing modes*.



32-bit (size == 10)

```
LDR <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

64-bit (size == 11)

```
LDR <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount> For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#2

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#3

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, datasize DIV 8, AccType_NORMAL];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDRAA, LDRAB

Load Register, with pointer authentication. This instruction authenticates an address from a base register using a modifier of zero and the specified key, adds an immediate offset to the authenticated address, and loads a 64-bit doubleword from memory at this resulting address into a register.

Key A is used for LDRAA, and key B is used for LDRAB.

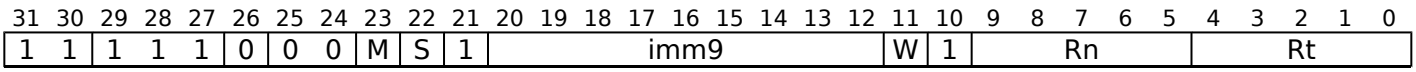
If the authentication passes, the PE behaves the same as for an LDR instruction. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to the base register, unless the pre-indexed variant of the instruction is used. In this case, the address that is written back to the base register does not include the pointer authentication code.

For information about memory accesses, see *Load/Store addressing modes*.

Unscaled offset

(FEAT_PAuth)



Key A, offset (M == 0 && W == 0)

```
LDRAA <Xt>, [<Xn|SP>{, #<simm>}]
```

Key A, pre-indexed (M == 0 && W == 1)

```
LDRAA <Xt>, [<Xn|SP>{, #<simm>}]!
```

Key B, offset (M == 1 && W == 0)

```
LDRAB <Xt>, [<Xn|SP>{, #<simm>}]
```

Key B, pre-indexed (M == 1 && W == 1)

```
LDRAB <Xt>, [<Xn|SP>{, #<simm>}]!
```

```
if !HavePACExt() then UNDEFINED;
integer t = UInt(Rt);
integer n = UInt(Rn);
boolean wback = (W == '1');
boolean use_key_a = (M == '0');
bits(10) S10 = S:imm9;
bits(64) offset = LSL(SignExtend(S10, 64), 3);
boolean tag_checked = wback || n != 31;
```

Assembler Symbols

<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, a multiple of 8 in the range -4096 to 4088, defaulting to 0 and encoded in the "S:imm9" field as <simm>/8.

Operation

```
bits(64) address;
bits(64) data;
boolean wb_unknown = FALSE;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if n == 31 then
    address = SP[];
else
    address = X[n, 64];

if use_key_a then
    address = AuthDA(address, X[31, 64], TRUE);
else
    address = AuthDB(address, X[31, 64], TRUE);

if n == 31 then
    CheckSPAlignment();

address = address + offset;
data = Mem[address, 8, AccType_NORMAL];
X[t, 64] = data;

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

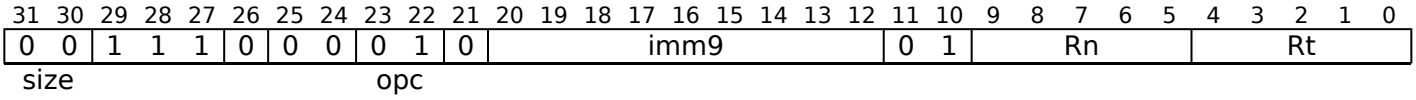
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDRB (immediate)

Load Register Byte (immediate) loads a byte from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

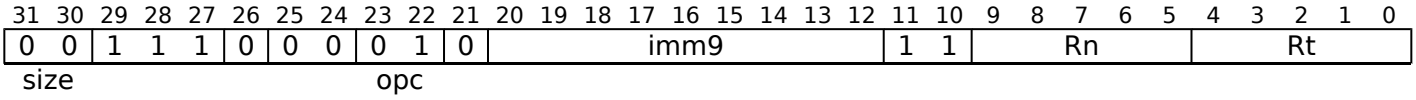
Post-index



LDRB <Wt>, [<Xn|SP>], #<simm>

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

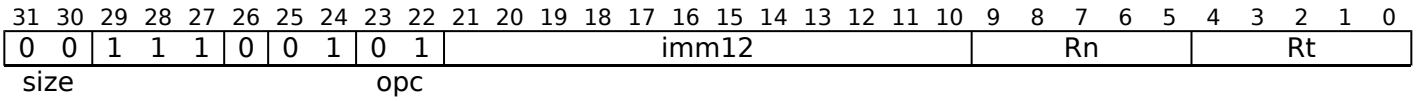
Pre-index



LDRB <Wt>, [<Xn|SP>, #<simm>]!

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



LDRB <Wt>, [<Xn|SP>{, #<pimm>}]

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDRH (immediate)*.

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = wback || n != 31;

boolean wb_unknown = FALSE;
Constraint c;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();
```

Operation

```
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

data = Mem[address, 1, AccType_NORMAL];
X[t, 32] = ZeroExtend(data, 32);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

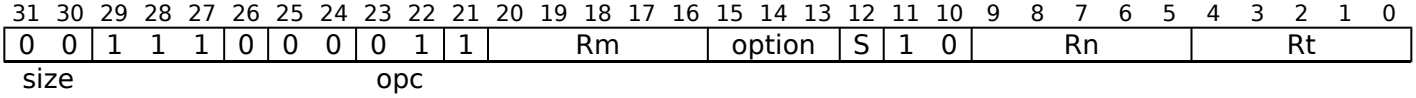
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDRB (register)

Load Register Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.



Extended register (option != 011)

```
LDRB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

Shifted register (option == 011)

```
LDRB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

```
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend specifier, encoded in “option”:

option	<extend>
010	UXTW
110	SXTW
111	SXTX

<amount> Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, 0, 64);
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, 1, AccType_NORMAL];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

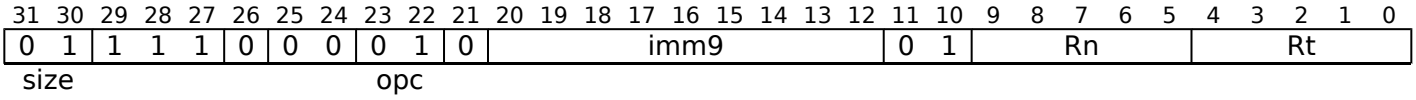
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDRH (immediate)

Load Register Halfword (immediate) loads a halfword from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

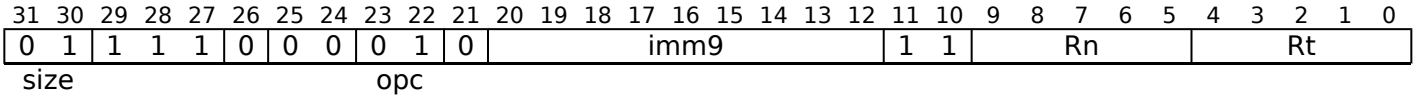
Post-index



LDRH <Wt>, [<Xn|SP>], #<simm>

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

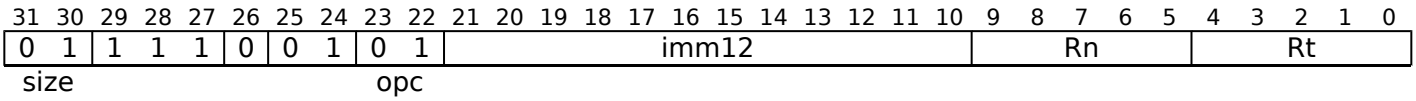
Pre-index



LDRH <Wt>, [<Xn|SP>, #<simm>]!

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



LDRH <Wt>, [<Xn|SP>{, #<pimm>}]

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDRH (immediate)*.

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = wback || n != 31;

boolean wb_unknown = FALSE;
Constraint c;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();
```

Operation

```
bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

data = Mem[address, 2, AccType_NORMAL];
X[t, 32] = ZeroExtend(data, 32);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

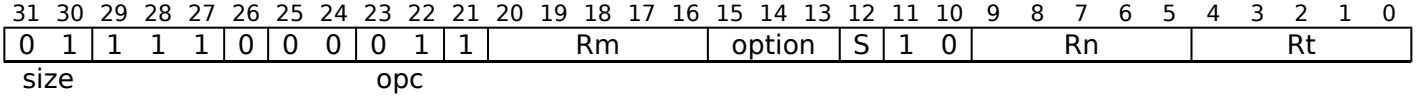
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDRH (register)

Load Register Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.



LDRH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

```
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 1 else 0;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#1

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, 2, AccType_NORMAL];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

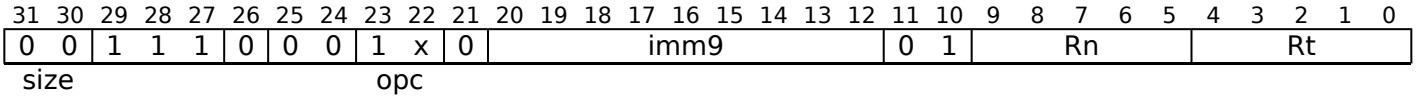
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDRSB (immediate)

Load Register Signed Byte (immediate) loads a byte from memory, sign-extends it to either 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index



32-bit (opc == 11)

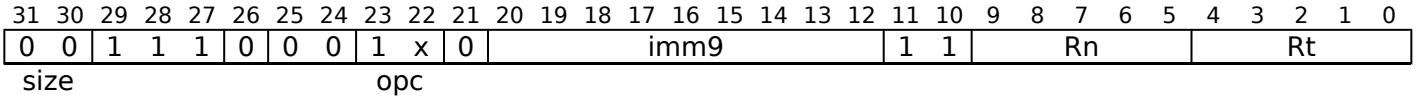
```
LDRSB <Wt>, [<Xn|SP>], #<simm>
```

64-bit (opc == 10)

```
LDRSB <Xt>, [<Xn|SP>], #<simm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index



32-bit (opc == 11)

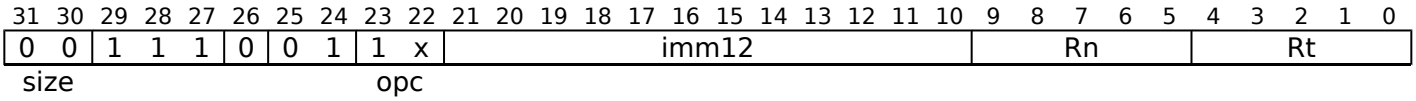
```
LDRSB <Wt>, [<Xn|SP>, #<simm>]!
```

64-bit (opc == 10)

```
LDRSB <Xt>, [<Xn|SP>, #<simm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



32-bit (opc == 11)

```
LDRSB <Wt>, [<Xn|SP>{, #<pimm>}]
```

64-bit (opc == 10)

```
LDRSB <Xt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDRSB (immediate)*.

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;
Constraint c;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Operation

```
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(8) UNKNOWN;
        else
            data = X[t, 8];
        Mem[address, 1, AccType_NORMAL] = data;

    when MemOp_LOAD
        data = Mem[address, 1, AccType_NORMAL];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

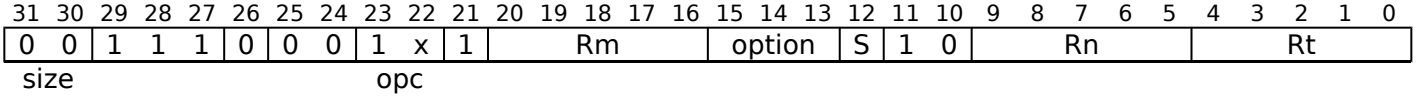
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDRSB (register)

Load Register Signed Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.



32-bit with extended register offset (opc == 11 && option != 011)

```
LDRSB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

32-bit with shifted register offset (opc == 11 && option == 011)

```
LDRSB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

64-bit with extended register offset (opc == 10 && option != 011)

```
LDRSB <Xt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

64-bit with shifted register offset (opc == 10 && option == 011)

```
LDRSB <Xt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

```
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend specifier, encoded in “option”:

option	<extend>
010	UXTW
110	SXTW
111	SXTX

<amount> Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, 0, 64);
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t, 8];
        Mem[address, 1, AccType_NORMAL] = data;

    when MemOp_LOAD
        data = Mem[address, 1, AccType_NORMAL];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

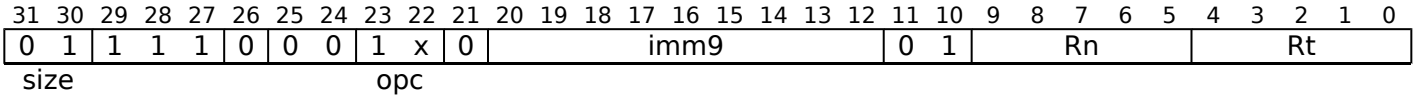
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDRSH (immediate)

Load Register Signed Halfword (immediate) loads a halfword from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index



32-bit (opc == 11)

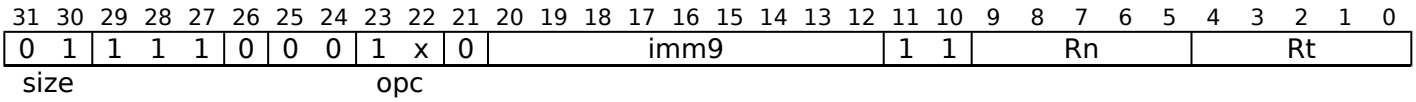
```
LDRSH <Wt>, [<Xn|SP>], #<simm>
```

64-bit (opc == 10)

```
LDRSH <Xt>, [<Xn|SP>], #<simm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index



32-bit (opc == 11)

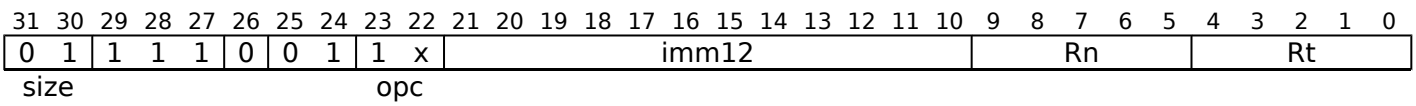
```
LDRSH <Wt>, [<Xn|SP>, #<simm>]!
```

64-bit (opc == 10)

```
LDRSH <Xt>, [<Xn|SP>, #<simm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



32-bit (opc == 11)

```
LDRSH <Wt>, [<Xn|SP>{, #<pimm>}]
```

64-bit (opc == 10)

```
LDRSH <Xt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDRSH (immediate)*.

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;
Constraint c;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Operation

```
bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        if rt_unknown then
            data = bits(16) UNKNOWN;
        else
            data = X[t, 16];
        Mem[address, 2, AccType_NORMAL] = data;

    when MemOp_LOAD
        data = Mem[address, 2, AccType_NORMAL];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

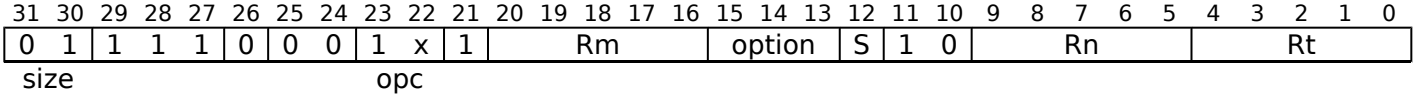
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDRSH (register)

Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it, and writes it to a register. For information about memory accesses see *Load/Store addressing modes*.



32-bit (opc == 11)

```
LDRSH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

64-bit (opc == 10)

```
LDRSH <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 1 else 0;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#1

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t, 16];
        Mem[address, 2, AccType_NORMAL] = data;

    when MemOp_LOAD
        data = Mem[address, 2, AccType_NORMAL];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

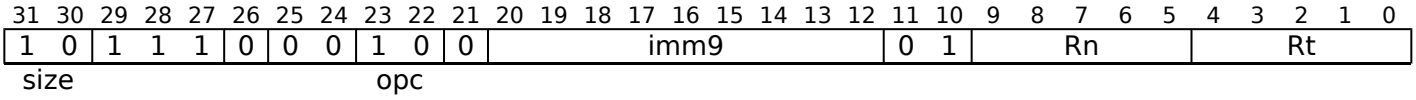
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDRSW (immediate)

Load Register Signed Word (immediate) loads a word from memory, sign-extends it to 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

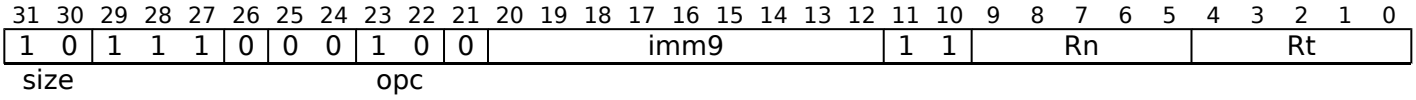
Post-index



LDRSW <Xt>, [<Xn|SP>], #<simm>

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

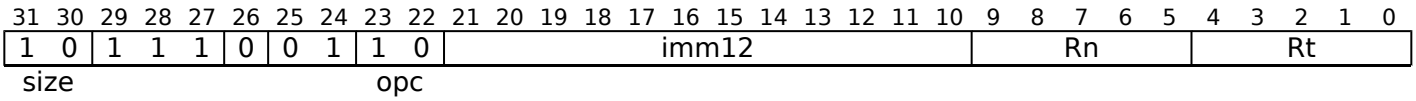
Pre-index



LDRSW <Xt>, [<Xn|SP>, #<simm>]!

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



LDRSW <Xt>, [<Xn|SP>{, #<pimm>}]

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 2);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDRSW (immediate)*.

Assembler Symbols

<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	Is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = wback || n != 31;

boolean wb_unknown = FALSE;
Constraint c;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;    // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();
```

Operation

```
bits(64) address;
bits(32) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

data = Mem[address, 4, AccType_NORMAL];
X[t, 64] = SignExtend(data, 64);
if wback then
    if wb_unknown then
        address = bits(64) UNKNOWN;
    elsif postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

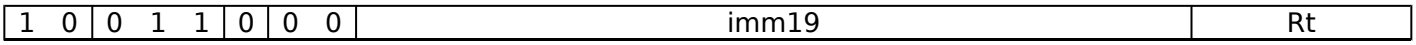
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDRSW (literal)

Load Register Signed Word (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.



opc

LDRSW <Xt>, <label>

```
integer t = UInt(Rt);
bits(64) offset;

offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<label>	Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```
bits(64) address = PC[] + offset;
bits(32) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);

data = Mem[address, 4, AccType_NORMAL];
X[t, 64] = SignExtend(data, 64);
```

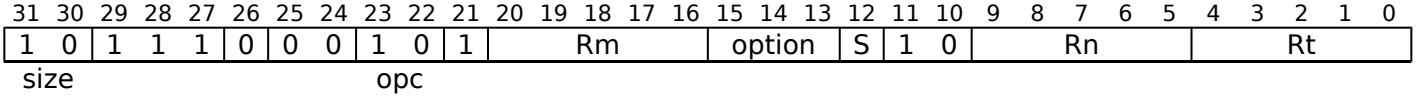
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDRSW (register)

Load Register Signed Word (register) calculates an address from a base register value and an offset register value, loads a word from memory, sign-extends it to form a 64-bit value, and writes it to a register. The offset register value can be shifted left by 0 or 2 bits. For information about memory accesses, see *Load/Store addressing modes*.



LDRSW <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]

```
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 2 else 0;
```

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#2

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;
bits(32) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, 4, AccType_NORMAL];
X[t, 64] = SignExtend(data, 64);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDSET, LDSETA, LDSETAL, LDSETL

Atomic bit set on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, `LDSETA` and `LDSETAL` load from memory with acquire semantics.
- `LDSETL` and `LDSETAL` store to memory with release semantics.
- `LDSET` has neither acquire nor release semantics.

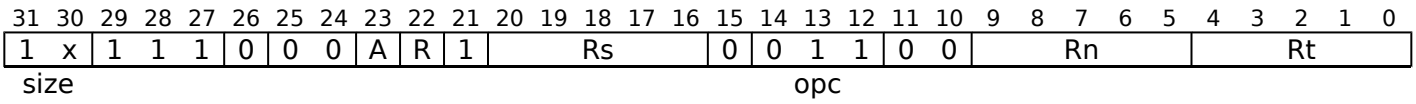
For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STSET, STSETL.

Integer

(FEAT_LSE)



32-bit LDSET (size == 10 && A == 0 && R == 0)

```
LDSET <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSETA (size == 10 && A == 1 && R == 0)

```
LDSETA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSETAL (size == 10 && A == 1 && R == 1)

```
LDSETAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSETL (size == 10 && A == 0 && R == 1)

```
LDSETL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDSET (size == 11 && A == 0 && R == 0)

```
LDSET <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSETA (size == 11 && A == 1 && R == 0)

```
LDSETA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSETAL (size == 11 && A == 1 && R == 1)

```
LDSETAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSETL (size == 11 && A == 0 && R == 1)

```
LDSETL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STSET, STSETL	`A == '0' && Rt == '11111'`

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, datasize];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_ORR, value, ldacctype, stacctype);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDSETB, LDSETAB, LDSETALB, LDSETLB

Atomic bit set on byte in memory atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSETAB and LDSETALB load from memory with acquire semantics.
- LDSETLB and LDSETALB store to memory with release semantics.
- LDSETB has neither acquire nor release semantics.

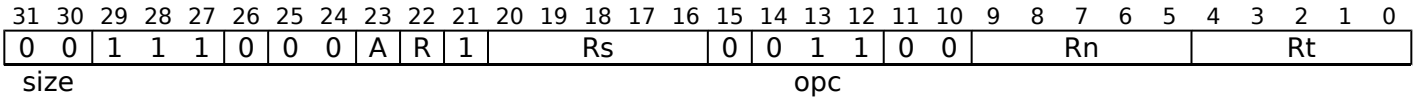
For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STSETB, STSETLB.

Integer

(FEAT_LSE)



LDSETAB (A == 1 && R == 0)

```
LDSETAB <Ws>, <Wt>, [<Xn|SP>]
```

LDSETALB (A == 1 && R == 1)

```
LDSETALB <Ws>, <Wt>, [<Xn|SP>]
```

LDSETB (A == 0 && R == 0)

```
LDSETB <Ws>, <Wt>, [<Xn|SP>]
```

LDSETLB (A == 0 && R == 1)

```
LDSETLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STSETB, STSETLB	`A == '0' && Rt == '11111'`

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, 8];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_ORR, value, ldacctype, stacctype);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDSETH, LDSETAH, LDSETALH, LDSETLH

Atomic bit set on halfword in memory atomically loads a 16-bit halfword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSETAH and LDSETALH load from memory with acquire semantics.
- LDSETLH and LDSETALH store to memory with release semantics.
- LDSETH has neither acquire nor release semantics.

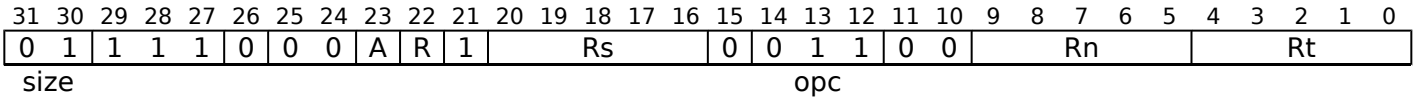
For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STSETH, STSETLH.

Integer

(FEAT_LSE)



LDSETAH (A == 1 && R == 0)

```
LDSETAH <Ws>, <Wt>, [<Xn|SP>]
```

LDSETALH (A == 1 && R == 1)

```
LDSETALH <Ws>, <Wt>, [<Xn|SP>]
```

LDSETH (A == 0 && R == 0)

```
LDSETH <Ws>, <Wt>, [<Xn|SP>]
```

LDSETLH (A == 0 && R == 1)

```
LDSETLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STSETH, STSETLH	`A == '0' && Rt == '11111'`

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, 16];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_ORR, value, ldacctype, stacctype);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL

Atomic signed maximum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, `LDSMAXA` and `LDSMAXAL` load from memory with acquire semantics.
- `LDSMAXL` and `LDSMAXAL` store to memory with release semantics.
- `LDSMAX` has neither acquire nor release semantics.

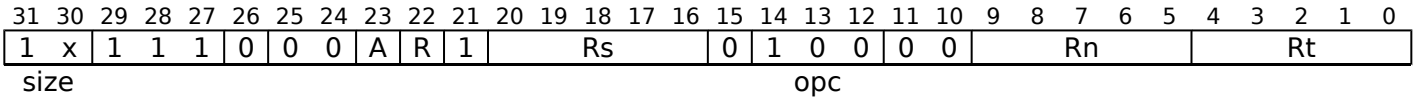
For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STSMAX, STSMAXL.

Integer

(FEAT_LSE)



32-bit LDSMAX (size == 10 && A == 0 && R == 0)

```
LDSMAX <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSMAXA (size == 10 && A == 1 && R == 0)

```
LDSMAXA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSMAXAL (size == 10 && A == 1 && R == 1)

```
LDSMAXAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSMAXL (size == 10 && A == 0 && R == 1)

```
LDSMAXL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDSMAX (size == 11 && A == 0 && R == 0)

```
LDSMAX <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSMAXA (size == 11 && A == 1 && R == 0)

```
LDSMAXA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSMAXAL (size == 11 && A == 1 && R == 1)

```
LDSMAXAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSMAXL (size == 11 && A == 0 && R == 1)

```
LDSMAXL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STSMAX, STSMAXL	`A == '0' && Rt == '11111'`

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, datasize];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_SMAX, value, ldacctype, stacctype);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB

Atomic signed maximum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

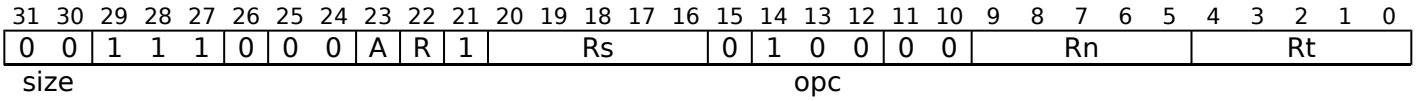
- If the destination register is not WZR, LDSMAXAB and LDSMAXALB load from memory with acquire semantics.
- LDSMAXLB and LDSMAXALB store to memory with release semantics.
- LDSMAXB has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STSMAXB, STSMAXLB.

Integer
(FEAT_LSE)



LDSMAXAB (A == 1 && R == 0)

```
LDSMAXAB <Ws>, <Wt>, [<Xn|SP>]
```

LDSMAXALB (A == 1 && R == 1)

```
LDSMAXALB <Ws>, <Wt>, [<Xn|SP>]
```

LDSMAXB (A == 0 && R == 0)

```
LDSMAXB <Ws>, <Wt>, [<Xn|SP>]
```

LDSMAXLB (A == 0 && R == 1)

```
LDSMAXLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STSMAXB, STSMAXLB	`A == '0' && Rt == '11111'`

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, 8];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_SMAX, value, ldacctype, stacctype);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH

Atomic signed maximum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMAXAH and LDSMAXALH load from memory with acquire semantics.
- LDSMAXLH and LDSMAXALH store to memory with release semantics.
- LDSMAXH has neither acquire nor release semantics.

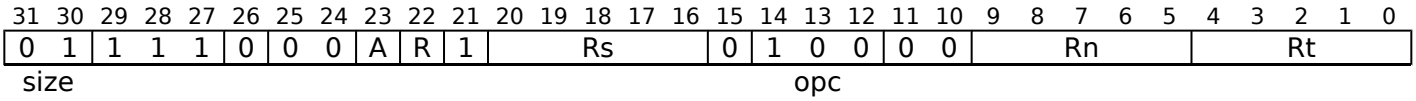
For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STSMAXH, STSMAXLH.

Integer

(FEAT_LSE)



LDSMAXAH (A == 1 && R == 0)

```
LDSMAXAH <Ws>, <Wt>, [<Xn|SP>]
```

LDSMAXALH (A == 1 && R == 1)

```
LDSMAXALH <Ws>, <Wt>, [<Xn|SP>]
```

LDSMAXH (A == 0 && R == 0)

```
LDSMAXH <Ws>, <Wt>, [<Xn|SP>]
```

LDSMAXLH (A == 0 && R == 1)

```
LDSMAXLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STSMAXH, STSMAXLH	`A == '0' && Rt == '11111'`

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, 16];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_SMAX, value, ldacctype, stacctype);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDSMIN, LDSMINA, LDSMINAL, LDSMINL

Atomic signed minimum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, `LDSMINA` and `LDSMINAL` load from memory with acquire semantics.
- `LDSMINL` and `LDSMINAL` store to memory with release semantics.
- `LDSMIN` has neither acquire nor release semantics.

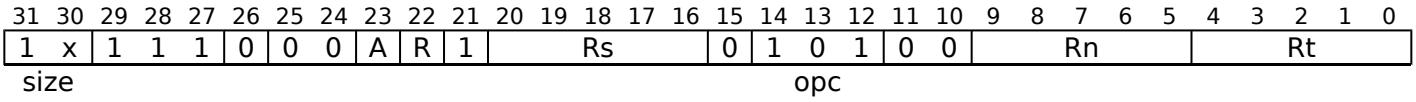
For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STSMIN, STSMINL.

Integer

(FEAT_LSE)



32-bit LDSMIN (size == 10 && A == 0 && R == 0)

```
LDSMIN <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSMINA (size == 10 && A == 1 && R == 0)

```
LDSMINA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSMINAL (size == 10 && A == 1 && R == 1)

```
LDSMINAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDSMINL (size == 10 && A == 0 && R == 1)

```
LDSMINL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDSMIN (size == 11 && A == 0 && R == 0)

```
LDSMIN <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSMINA (size == 11 && A == 1 && R == 0)

```
LDSMINA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSMINAL (size == 11 && A == 1 && R == 1)

```
LDSMINAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDSMINL (size == 11 && A == 0 && R == 1)

```
LDSMINL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STSMIN, STSMINL	`A == '0' && Rt == '11111'`

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, datasize];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_SMIN, value, ldacctype, stacctype);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB

Atomic signed minimum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMINAB and LDSMINALB load from memory with acquire semantics.
- LDSMINLB and LDSMINALB store to memory with release semantics.
- LDSMINB has neither acquire nor release semantics.

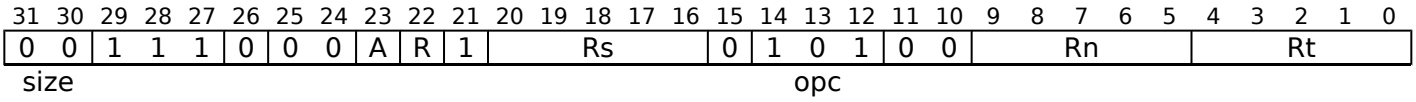
For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STSMINB, STSMINLB.

Integer

(FEAT_LSE)



LDSMINAB (A == 1 && R == 0)

```
LDSMINAB <Ws>, <Wt>, [<Xn|SP>]
```

LDSMINALB (A == 1 && R == 1)

```
LDSMINALB <Ws>, <Wt>, [<Xn|SP>]
```

LDSMINB (A == 0 && R == 0)

```
LDSMINB <Ws>, <Wt>, [<Xn|SP>]
```

LDSMINLB (A == 0 && R == 1)

```
LDSMINLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STSMINB, STSMINLB	`A == '0' && Rt == '11111'`

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, 8];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_SMIN, value, ldacctype, stacctype);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH

Atomic signed minimum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMINAH and LDSMINALH load from memory with acquire semantics.
- LDSMINLH and LDSMINALH store to memory with release semantics.
- LDSMINH has neither acquire nor release semantics.

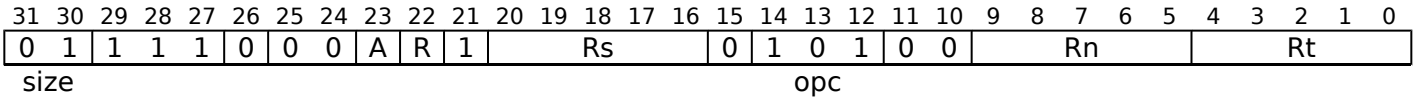
For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STSMINH, STSMINLH.

Integer

(FEAT_LSE)



LDSMINAH (A == 1 && R == 0)

```
LDSMINAH <Ws>, <Wt>, [<Xn|SP>]
```

LDSMINALH (A == 1 && R == 1)

```
LDSMINALH <Ws>, <Wt>, [<Xn|SP>]
```

LDSMINH (A == 0 && R == 0)

```
LDSMINH <Ws>, <Wt>, [<Xn|SP>]
```

LDSMINLH (A == 0 && R == 1)

```
LDSMINLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STSMINH, STSMINLH	A == '0' && Rt == '11111'

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, 16];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_SMIN, value, ldacctype, stacctype);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

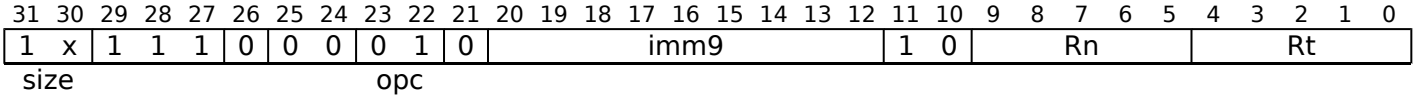
LDTR

Load Register (unprivileged) loads a word or doubleword from memory, and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of *HCR_EL2*.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.



32-bit (size == 10)

```
LDTR <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit (size == 11)

```
LDTR <Xt>, [<Xn|SP>{, #<simm>}]
```

```
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype;
unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tag_checked = n != 31;
```

Operation

```
bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, datasize DIV 8, acctype];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

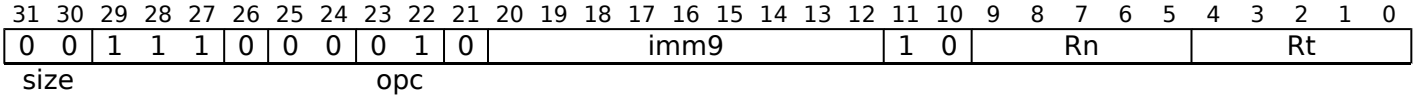
LDTRB

Load Register Byte (unprivileged) loads a byte from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of *HCR_EL2*.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.



LDTRB <Wt>, [<Xn|SP>{, #<simm>}]

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype;
unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

boolean tag_checked = n != 31;
```

Operation

```
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, 1, acctype];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

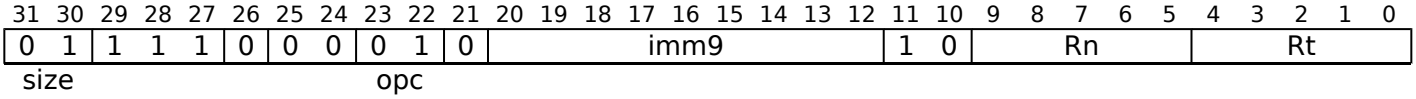
LDTRH

Load Register Halfword (unprivileged) loads a halfword from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of *HCR_EL2*.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.



LDTRH <Wt>, [<Xn|SP>{, #<simm>}]

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype;
unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

boolean tag_checked = n != 31;
```

Operation

```
bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, 2, acctype];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

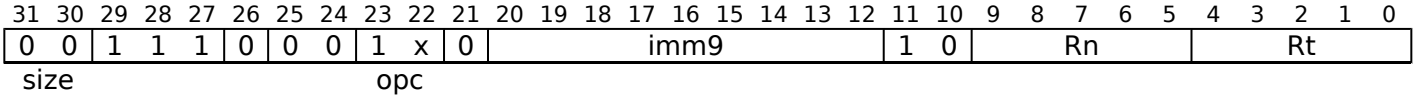
LDTRSB

Load Register Signed Byte (unprivileged) loads a byte from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of *HCR_EL2*.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.



32-bit (opc == 11)

```
LDTRSB <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit (opc == 10)

```
LDTRSB <Xt>, [<Xn|SP>{, #<simm>}]

bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype;
unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);
```

Operation

```
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t, 8];
        Mem[address, 1, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, 1, acctype];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

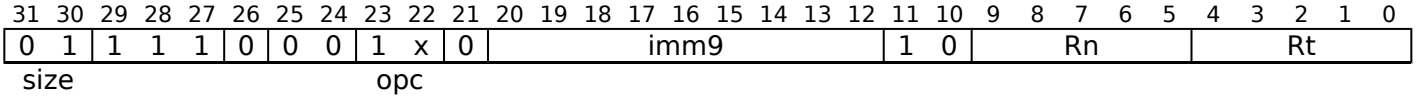
LDTRSH

Load Register Signed Halfword (unprivileged) loads a halfword from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of *HCR_EL2*.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.



32-bit (opc == 11)

```
LDTRSH <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit (opc == 10)

```
LDTRSH <Xt>, [<Xn|SP>{, #<simm>}]

bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype;
unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);
```

Operation

```
bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t, 16];
        Mem[address, 2, acctype] = data;

    when MemOp_LOAD
        data = Mem[address, 2, acctype];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

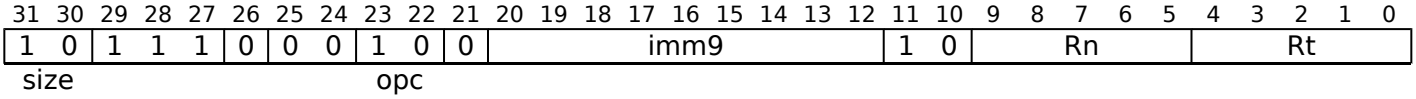
LDTRSW

Load Register Signed Word (unprivileged) loads a word from memory, sign-extends it to 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of *HCR_EL2*.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.



LDTRSW <Xt>, [<Xn|SP>{, #<simm>}]

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype;
unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

boolean tag_checked = n != 31;
```

Operation

```
bits(64) address;
bits(32) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, 4, acctype];
X[t, 64] = SignExtend(data, 64);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL

Atomic unsigned maximum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, `LDUMAXA` and `LDUMAXAL` load from memory with acquire semantics.
- `LDUMAXL` and `LDUMAXAL` store to memory with release semantics.
- `LDUMAX` has neither acquire nor release semantics.

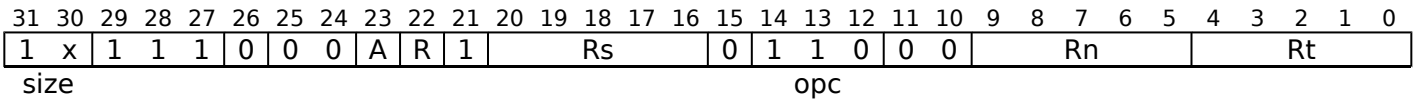
For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STUMAX, STUMAXL.

Integer

(FEAT_LSE)



32-bit LDUMAX (size == 10 && A == 0 && R == 0)

```
LDUMAX <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDUMAXA (size == 10 && A == 1 && R == 0)

```
LDUMAXA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDUMAXAL (size == 10 && A == 1 && R == 1)

```
LDUMAXAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDUMAXL (size == 10 && A == 0 && R == 1)

```
LDUMAXL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDUMAX (size == 11 && A == 0 && R == 0)

```
LDUMAX <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDUMAXA (size == 11 && A == 1 && R == 0)

```
LDUMAXA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDUMAXAL (size == 11 && A == 1 && R == 1)

```
LDUMAXAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDUMAXL (size == 11 && A == 0 && R == 1)

```
LDUMAXL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STUMAX, STUMAXL	`A == '0' && Rt == '11111'`

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, datasize];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_UMAX, value, ldacctype, stacctype);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB

Atomic unsigned maximum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMAXAB and LDUMAXALB load from memory with acquire semantics.
- LDUMAXLB and LDUMAXALB store to memory with release semantics.
- LDUMAXB has neither acquire nor release semantics.

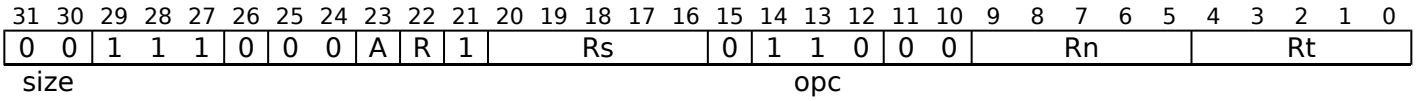
For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STUMAXB, STUMAXLB.

Integer

(FEAT_LSE)



LDUMAXAB (A == 1 && R == 0)

```
LDUMAXAB <Ws>, <Wt>, [<Xn|SP>]
```

LDUMAXALB (A == 1 && R == 1)

```
LDUMAXALB <Ws>, <Wt>, [<Xn|SP>]
```

LDUMAXB (A == 0 && R == 0)

```
LDUMAXB <Ws>, <Wt>, [<Xn|SP>]
```

LDUMAXLB (A == 0 && R == 1)

```
LDUMAXLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STUMAXB, STUMAXLB	`A == '0' && Rt == '11111'`

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, 8];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_UMAX, value, ldacctype, stacctype);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH

Atomic unsigned maximum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMAXAH and LDUMAXALH load from memory with acquire semantics.
- LDUMAXLH and LDUMAXALH store to memory with release semantics.
- LDUMAXH has neither acquire nor release semantics.

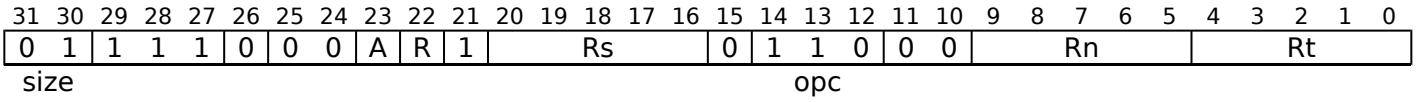
For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STUMAXH, STUMAXLH.

Integer

(FEAT_LSE)



LDUMAXAH (A == 1 && R == 0)

```
LDUMAXAH <Ws>, <Wt>, [<Xn|SP>]
```

LDUMAXALH (A == 1 && R == 1)

```
LDUMAXALH <Ws>, <Wt>, [<Xn|SP>]
```

LDUMAXH (A == 0 && R == 0)

```
LDUMAXH <Ws>, <Wt>, [<Xn|SP>]
```

LDUMAXLH (A == 0 && R == 1)

```
LDUMAXLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STUMAXH, STUMAXLH	`A == '0' && Rt == '11111'`

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, 16];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_UMAX, value, ldacctype, stacctype);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDUMIN, LDUMINA, LDUMINAL, LDUMINL

Atomic unsigned minimum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, `LDUMINA` and `LDUMINAL` load from memory with acquire semantics.
- `LDUMINL` and `LDUMINAL` store to memory with release semantics.
- `LDUMIN` has neither acquire nor release semantics.

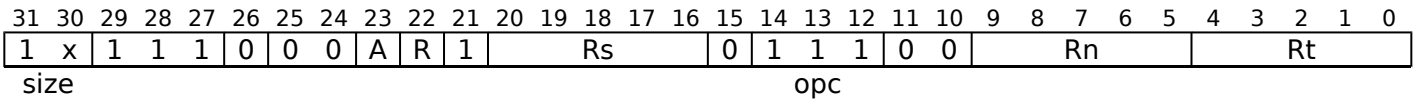
For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STUMIN, STUMINL.

Integer

(FEAT_LSE)



32-bit LDUMIN (size == 10 && A == 0 && R == 0)

```
LDUMIN <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDUMINA (size == 10 && A == 1 && R == 0)

```
LDUMINA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDUMINAL (size == 10 && A == 1 && R == 1)

```
LDUMINAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit LDUMINL (size == 10 && A == 0 && R == 1)

```
LDUMINL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit LDUMIN (size == 11 && A == 0 && R == 0)

```
LDUMIN <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDUMINA (size == 11 && A == 1 && R == 0)

```
LDUMINA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDUMINAL (size == 11 && A == 1 && R == 1)

```
LDUMINAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit LDUMINL (size == 11 && A == 0 && R == 1)

```
LDUMINL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STUMIN, STUMINL	`A == '0' && Rt == '11111'`

Operation

```
bits(64) address;
bits(datasize) value;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, datasize];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_UMIN, value, ldacctype, stacctype);

if t != 31 then
    X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB

Atomic unsigned minimum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, `LDUMINAB` and `LDUMINALB` load from memory with acquire semantics.
- `LDUMINLB` and `LDUMINALB` store to memory with release semantics.
- `LDUMINB` has neither acquire nor release semantics.

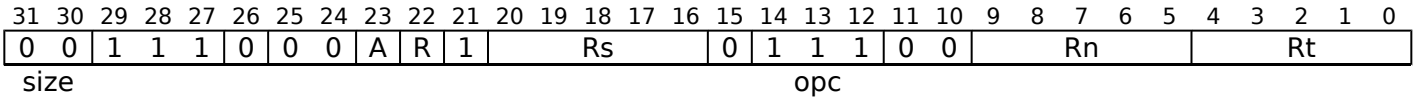
For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STUMINB, STUMINLB.

Integer

(FEAT_LSE)



LDUMINAB (A == 1 && R == 0)

```
LDUMINAB <Ws>, <Wt>, [<Xn|SP>]
```

LDUMINALB (A == 1 && R == 1)

```
LDUMINALB <Ws>, <Wt>, [<Xn|SP>]
```

LDUMINB (A == 0 && R == 0)

```
LDUMINB <Ws>, <Wt>, [<Xn|SP>]
```

LDUMINLB (A == 0 && R == 1)

```
LDUMINLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STUMINB, STUMINLB	`A == '0' && Rt == '11111'`

Operation

```
bits(64) address;
bits(8) value;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, 8];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_UMIN, value, ldacctype, stacctype);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH

Atomic unsigned minimum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMINAH and LDUMINALH load from memory with acquire semantics.
- LDUMINLH and LDUMINALH store to memory with release semantics.
- LDUMINH has neither acquire nor release semantics.

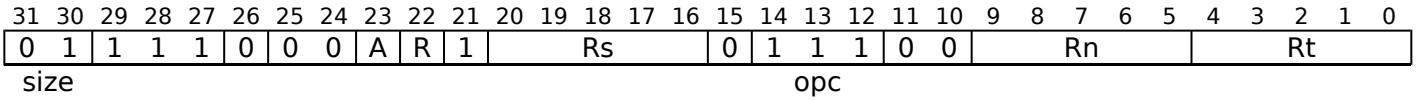
For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STUMINH, STUMINLH.

Integer

(FEAT_LSE)



LDUMINAH (A == 1 && R == 0)

```
LDUMINAH <Ws>, <Wt>, [<Xn|SP>]
```

LDUMINALH (A == 1 && R == 1)

```
LDUMINALH <Ws>, <Wt>, [<Xn|SP>]
```

LDUMINH (A == 0 && R == 0)

```
LDUMINH <Ws>, <Wt>, [<Xn|SP>]
```

LDUMINLH (A == 0 && R == 1)

```
LDUMINLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STUMINH, STUMINLH	`A == '0' && Rt == '11111'`

Operation

```
bits(64) address;
bits(16) value;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

value = X[s, 16];
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = MemAtomic(address, MemAtomicOp_UMIN, value, ldacctype, stacctype);

if t != 31 then
    X[t, 32] = ZeroExtend(data, 32);
```

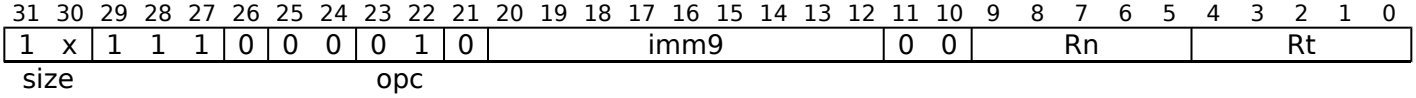
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDUR

Load Register (unscaled) calculates an address from a base register and an immediate offset, loads a 32-bit word or 64-bit doubleword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.



32-bit (size == 10)

```
LDUR <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit (size == 11)

```
LDUR <Xt>, [<Xn|SP>{, #<simm>}]
```

```
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer regsize;

regsize = if size == '11' then 64 else 32;
integer datasize = 8 << scale;
boolean tag_checked = n != 31;
```

Operation

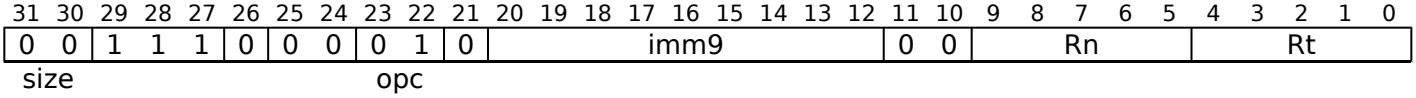
```
bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, datasize DIV 8, AccType_NORMAL];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

LDURB

Load Register Byte (unscaled) calculates an address from a base register and an immediate offset, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.



LDURB <Wt>, [<Xn|SP>{, #<simm>}]

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Operation

```
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, 1, AccType_NORMAL];
X[t, 32] = ZeroExtend(data, 32);
```

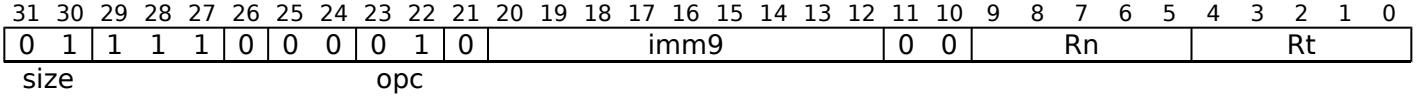
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDURH

Load Register Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a halfword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.



LDURH <Wt>, [<Xn|SP>{, #<simm>}]

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Operation

```
bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, 2, AccType_NORMAL];
X[t, 32] = ZeroExtend(data, 32);
```

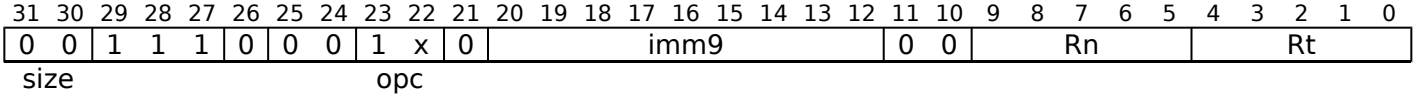
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDURSB

Load Register Signed Byte (unscaled) calculates an address from a base register and an immediate offset, loads a signed byte from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.



32-bit (opc == 11)

```
LDURSB <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit (opc == 10)

```
LDURSB <Xt>, [<Xn|SP>{, #<simm>}]
```

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);
```

Operation

```
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t, 8];
        Mem[address, 1, AccType_NORMAL] = data;

    when MemOp_LOAD
        data = Mem[address, 1, AccType_NORMAL];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

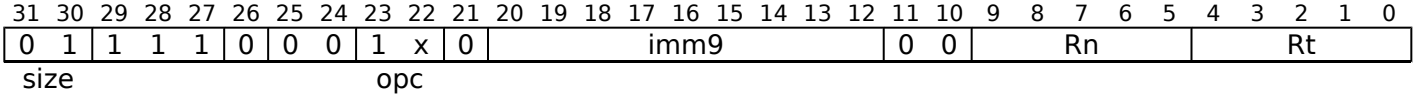
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDURSH

Load Register Signed Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a signed halfword from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.



32-bit (opc == 11)

```
LDURSH <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit (opc == 10)

```
LDURSH <Xt>, [<Xn|SP>{, #<simm>}]
```

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = 32;
    signed = FALSE;
else
    // sign-extending load
    memop = MemOp_LOAD;
    regsize = if opc<0> == '1' then 32 else 64;
    signed = TRUE;

boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);
```

Operation

```
bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    if memop != MemOp_PREFETCH then CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

case memop of
    when MemOp_STORE
        data = X[t, 16];
        Mem[address, 2, AccType_NORMAL] = data;

    when MemOp_LOAD
        data = Mem[address, 2, AccType_NORMAL];
        if signed then
            X[t, regsize] = SignExtend(data, regsize);
        else
            X[t, regsize] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

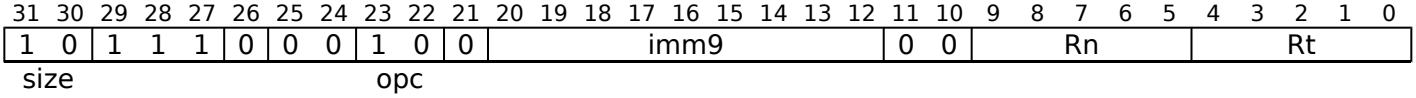
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDURSW

Load Register Signed Word (unscaled) calculates an address from a base register and an immediate offset, loads a signed word from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.



LDURSW <Xt>, [<Xn|SP>{, #<simm>}]

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Operation

```
bits(64) address;
bits(32) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = Mem[address, 4, AccType_NORMAL];
X[t, 64] = SignExtend(data, 64);
```

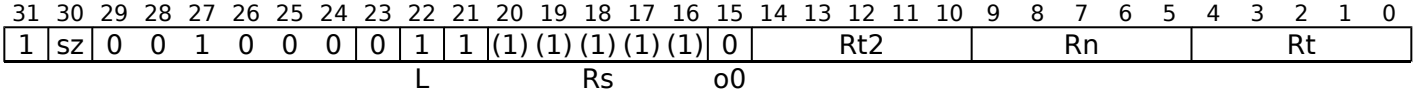
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDXP

Load Exclusive Pair of Registers derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information on single-copy atomicity and alignment requirements, see *Requirements for single-copy atomicity* and *Alignment of data accesses*. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. For information about memory accesses, see *Load/Store addressing modes*.



32-bit (sz == 0)

```
LDXP <Wt1>, <Wt2>, [<Xn|SP>{,#0}]
```

64-bit (sz == 1)

```
LDXP <Xt1>, <Xt2>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDXP*.

Assembler Symbols

<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

if rt_unknown then
    // ConstrainedUNPREDICTABLE case
    X[t, datasize] = bits(datasize) UNKNOWN;    // In this case t = t2
elsif elsize == 32 then
    // 32-bit load exclusive pair (atomic)
    data = Mem[address, dbytes, AccType_ATOMIC];
    if BigEndian(AccType_ATOMIC) then
        X[t, datasize-elsize] = data<datasize-1:elsize>;
        X[t2, elsize] = data<elsize-1:0>;
    else
        X[t, elsize] = data<elsize-1:0>;
        X[t2, datasize-elsize] = data<datasize-1:elsize>;
else // elsize == 64
    // 64-bit load exclusive pair (not atomic),
    // but must be 128-bit aligned
    if address != Align(address, dbytes) then
        AArch64.Abort(address, AlignmentFault(AccType_ATOMIC, FALSE, FALSE));
    X[t, 64] = Mem[address, 8, AccType_ATOMIC];
    X[t2, 64] = Mem[address+8, 8, AccType_ATOMIC];
```

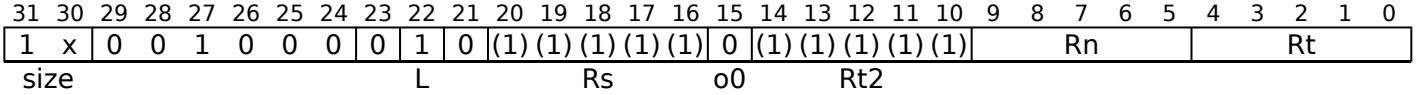
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDXR

Load Exclusive Register derives an address from a base register value, loads a 32-bit word or a 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. For information about memory accesses see *Load/Store addressing modes*.



32-bit (size == 10)

```
LDXR <Wt>, [<Xn|SP>{,#0}]
```

64-bit (size == 11)

```
LDXR <Xt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

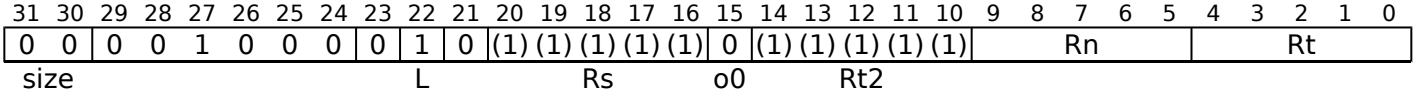
```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, dbytes);

data = Mem[address, dbytes, AccType_ATOMIC];
X[t, regsize] = ZeroExtend(data, regsize);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

LDXRB

Load Exclusive Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. For information about memory accesses see *Load/Store addressing modes*.



LDXRB <Wt>, [<Xn|SP>{,#0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 1);

data = Mem[address, 1, AccType_ATOMIC];
X[t, 32] = ZeroExtend(data, 32);
```

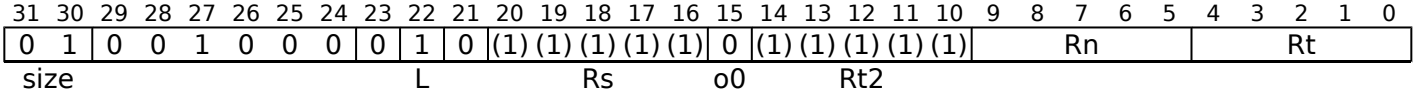
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDXRH

Load Exclusive Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. For information about memory accesses see *Load/Store addressing modes*.



LDXRH <Wt>, [<Xn|SP>{,#0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

// Tell the Exclusives monitors to record a sequence of one or more atomic
// memory reads from virtual address range [address, address+dbytes-1].
// The Exclusives monitor will only be set if all the reads are from the
// same dbytes-aligned physical address, to allow for the possibility of
// an atomicity break if the translation is changed between reads.
AArch64.SetExclusiveMonitors(address, 2);

data = Mem[address, 2, AccType_ATOMIC];
X[t, 32] = ZeroExtend(data, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

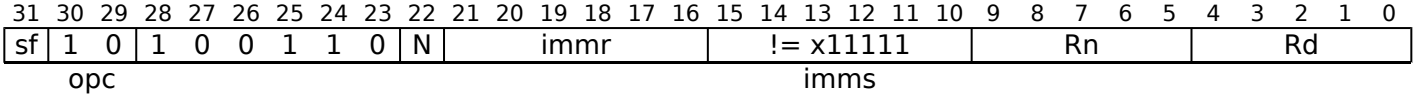
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LSL (immediate)

Logical Shift Left (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.



32-bit (sf == 0 && N == 0 && imms != 011111)

```
LSL <Wd>, <Wn>, #<shift>
```

is equivalent to

```
UBFM <Wd>, <Wn>, #(-<shift> MOD 32), #(31-<shift>)
```

and is the preferred disassembly when `imms + 1 == immr`.

64-bit (sf == 1 && N == 1 && imms != 111111)

```
LSL <Xd>, <Xn>, #<shift>
```

is equivalent to

```
UBFM <Xd>, <Xn>, #(-<shift> MOD 64), #(63-<shift>)
```

and is the preferred disassembly when `imms + 1 == immr`.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<shift>	For the 32-bit variant: is the shift amount, in the range 0 to 31. For the 64-bit variant: is the shift amount, in the range 0 to 63.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

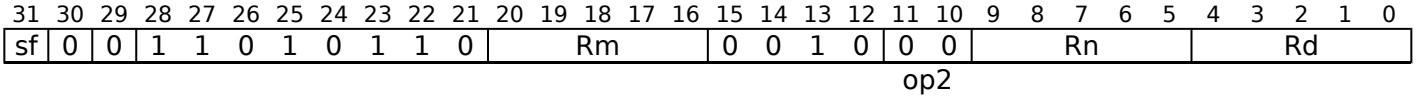
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LSL (register)

Logical Shift Left (register) shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is left-shifted.

This is an alias of LSLV. This means:

- The encodings in this description are named to match the encodings of LSLV.
- The description of LSLV gives the operational pseudocode for this instruction.



32-bit (sf == 0)

LSL <Wd>, <Wn>, <Wm>

is equivalent to

LSLV <Wd>, <Wn>, <Wm>

and is always the preferred disassembly.

64-bit (sf == 1)

LSL <Xd>, <Xn>, <Xm>

is equivalent to

LSLV <Xd>, <Xn>, <Xm>

and is always the preferred disassembly.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

The description of LSLV gives the operational pseudocode for this instruction.

Operational information

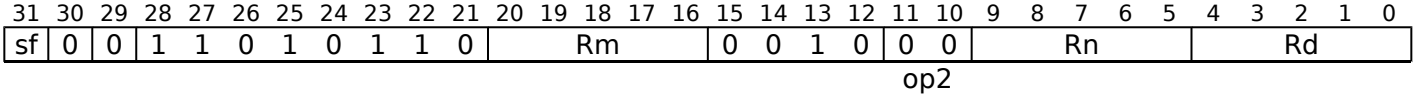
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

LSLV

Logical Shift Left Variable shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is left-shifted.

This instruction is used by the alias LSL (register).



32-bit (sf == 0)

```
LSLV <Wd>, <Wn>, <Wm>
```

64-bit (sf == 1)

```
LSLV <Xd>, <Xn>, <Xm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ShiftType shift_type = DecodeShift(op2);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

```
bits(datasize) result;
bits(datasize) operand2 = X[m, datasize];

result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize, datasize);
X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

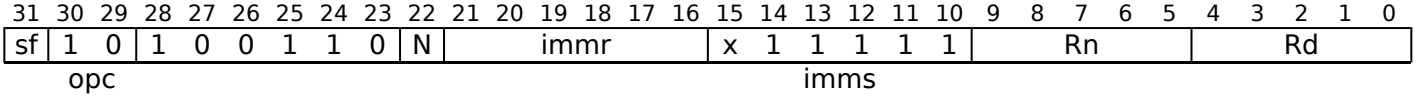
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LSR (immediate)

Logical Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.



32-bit (sf == 0 && N == 0 && imms == 011111)

```
LSR <Wd>, <Wn>, #<shift>
```

is equivalent to

```
UBFM <Wd>, <Wn>, #<shift>, #31
```

and is always the preferred disassembly.

64-bit (sf == 1 && N == 1 && imms == 111111)

```
LSR <Xd>, <Xn>, #<shift>
```

is equivalent to

```
UBFM <Xd>, <Xn>, #<shift>, #63
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<shift>	For the 32-bit variant: is the shift amount, in the range 0 to 31, encoded in the "immr" field. For the 64-bit variant: is the shift amount, in the range 0 to 63, encoded in the "immr" field.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

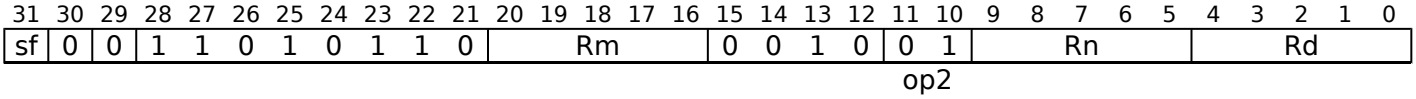
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LSR (register)

Logical Shift Right (register) shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This is an alias of LSRV. This means:

- The encodings in this description are named to match the encodings of LSRV.
- The description of LSRV gives the operational pseudocode for this instruction.



32-bit (sf == 0)

LSR <Wd>, <Wn>, <Wm>

is equivalent to

LSRV <Wd>, <Wn>, <Wm>

and is always the preferred disassembly.

64-bit (sf == 1)

LSR <Xd>, <Xn>, <Xm>

is equivalent to

LSRV <Xd>, <Xn>, <Xm>

and is always the preferred disassembly.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

The description of LSRV gives the operational pseudocode for this instruction.

Operational information

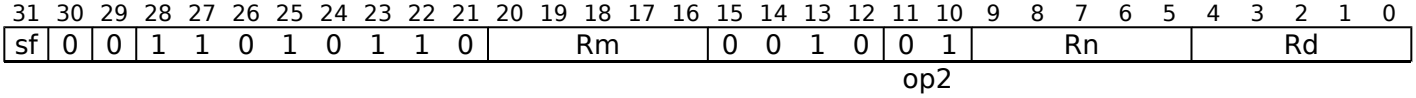
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

LSRV

Logical Shift Right Variable shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is used by the alias LSR (register).



32-bit (sf == 0)

```
LSRV <Wd>, <Wn>, <Wm>
```

64-bit (sf == 1)

```
LSRV <Xd>, <Xn>, <Xm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ShiftType shift_type = DecodeShift(op2);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

```
bits(datasize) result;
bits(datasize) operand2 = X[m, datasize];

result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize, datasize);
X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

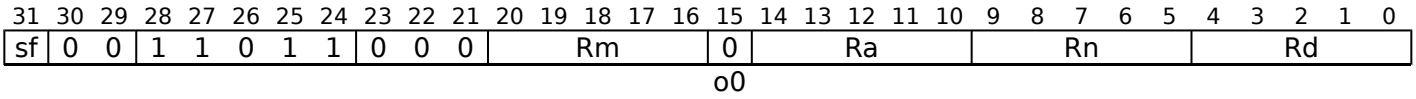
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MADD

Multiply-Add multiplies two register values, adds a third register value, and writes the result to the destination register.

This instruction is used by the alias MUL.



32-bit (sf == 0)

```
MADD <Wd>, <Wn>, <Wm>, <Wa>
```

64-bit (sf == 1)

```
MADD <Xd>, <Xn>, <Xm>, <Xa>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = if sf == '1' then 64 else 32;
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Wa>	Is the 32-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Xa>	Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

Alias Conditions

Alias	Is preferred when
MUL	`Ra == '11111'`

Operation

```
bits(destsize) operand1 = X[n, destsize];
bits(destsize) operand2 = X[m, destsize];
bits(destsize) operand3 = X[a, destsize];

integer result;

result = UInt(operand3) + (UInt(operand1) * UInt(operand2));

X[d, destsize] = result<destsize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

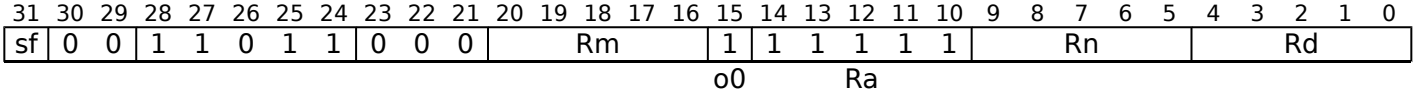
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MNEG

Multiply-Negate multiplies two register values, negates the product, and writes the result to the destination register.

This is an alias of MSUB. This means:

- The encodings in this description are named to match the encodings of MSUB.
- The description of MSUB gives the operational pseudocode for this instruction.



32-bit (sf == 0)

```
MNEG <Wd>, <Wn>, <Wm>
```

is equivalent to

```
MSUB <Wd>, <Wn>, <Wm>, WZR
```

and is always the preferred disassembly.

64-bit (sf == 1)

```
MNEG <Xd>, <Xn>, <Xm>
```

is equivalent to

```
MSUB <Xd>, <Xn>, <Xm>, XZR
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of MSUB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

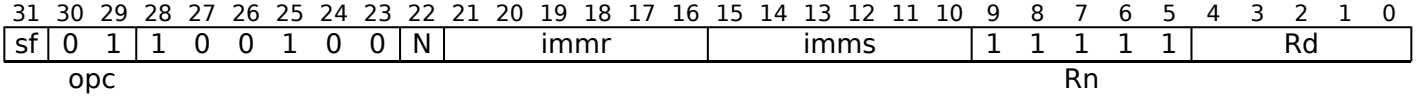
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

MOV (bitmask immediate)

Move (bitmask immediate) writes a bitmask immediate value to a register.

This is an alias of ORR (immediate). This means:

- The encodings in this description are named to match the encodings of ORR (immediate).
- The description of ORR (immediate) gives the operational pseudocode for this instruction.



32-bit (sf == 0 && N == 0)

```
MOV <Wd|WSP>, #<imm>
```

is equivalent to

```
ORR <Wd|WSP>, WZR, #<imm>
```

and is the preferred disassembly when `! MoveWidePreferred(sf, N, imms, immr)`.

64-bit (sf == 1)

```
MOV <Xd|SP>, #<imm>
```

is equivalent to

```
ORR <Xd|SP>, XZR, #<imm>
```

and is the preferred disassembly when `! MoveWidePreferred(sf, N, imms, immr)`.

Assembler Symbols

<Wd|WSP>	Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xd|SP>	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<imm>	For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr", but excluding values which could be encoded by MOVZ or MOVN. For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr", but excluding values which could be encoded by MOVZ or MOVN.

Operation

The description of ORR (immediate) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

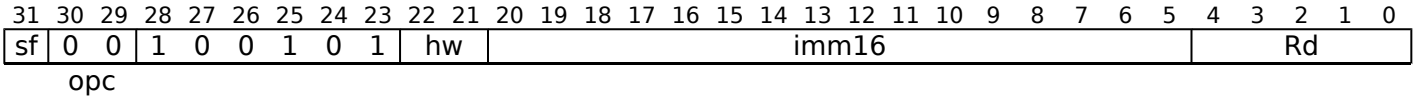
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MOV (inverted wide immediate)

Move (inverted wide immediate) moves an inverted 16-bit immediate value to a register.

This is an alias of MOVN. This means:

- The encodings in this description are named to match the encodings of MOVN.
- The description of MOVN gives the operational pseudocode for this instruction.



32-bit (sf == 0 && hw == 0x)

```
MOV <Wd>, #<imm>
```

is equivalent to

```
MOVN <Wd>, #<imm16>, LSL #<shift>
```

and is the preferred disassembly when `! (IsZero(imm16) && hw != '00') && ! IsOnes(imm16)`.

64-bit (sf == 1)

```
MOV <Xd>, #<imm>
```

is equivalent to

```
MOVN <Xd>, #<imm16>, LSL #<shift>
```

and is the preferred disassembly when `! (IsZero(imm16) && hw != '00')`.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<imm>	For the 32-bit variant: is a 32-bit immediate, the bitwise inverse of which can be encoded in "imm16:hw", but excluding 0xffff0000 and 0x0000ffff For the 64-bit variant: is a 64-bit immediate, the bitwise inverse of which can be encoded in "imm16:hw".
<shift>	For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16. For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Operation

The description of MOVN gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

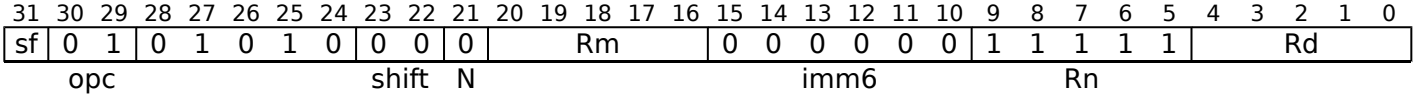
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

MOV (register)

Move (register) copies the value in a source register to the destination register.

This is an alias of ORR (shifted register). This means:

- The encodings in this description are named to match the encodings of ORR (shifted register).
- The description of ORR (shifted register) gives the operational pseudocode for this instruction.



32-bit (sf == 0)

```
MOV <Wd>, <Wm>
```

is equivalent to

```
ORR <Wd>, WZR, <Wm>
```

and is always the preferred disassembly.

64-bit (sf == 1)

```
MOV <Xd>, <Xm>
```

is equivalent to

```
ORR <Xd>, XZR, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wm>	Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xm>	Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

Operation

The description of ORR (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

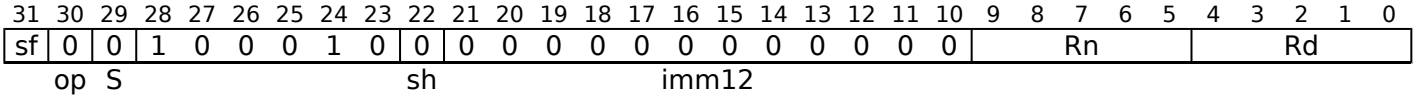
MOV (to/from SP)

Move between register and stack pointer

: `Rd = Rn`.

This is an alias of ADD (immediate). This means:

- The encodings in this description are named to match the encodings of ADD (immediate).
- The description of ADD (immediate) gives the operational pseudocode for this instruction.



32-bit (sf == 0)

```
MOV <Wd|WSP>, <Wn|WSP>
```

is equivalent to

```
ADD <Wd|WSP>, <Wn|WSP>, #0
```

and is the preferred disassembly when `(Rd == '11111' || Rn == '11111')`.

64-bit (sf == 1)

```
MOV <Xd|SP>, <Xn|SP>
```

is equivalent to

```
ADD <Xd|SP>, <Xn|SP>, #0
```

and is the preferred disassembly when `(Rd == '11111' || Rn == '11111')`.

Assembler Symbols

<Wd|WSP>	Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Wn|WSP>	Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<Xd|SP>	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xn|SP>	Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.

Operation

The description of ADD (immediate) gives the operational pseudocode for this instruction.

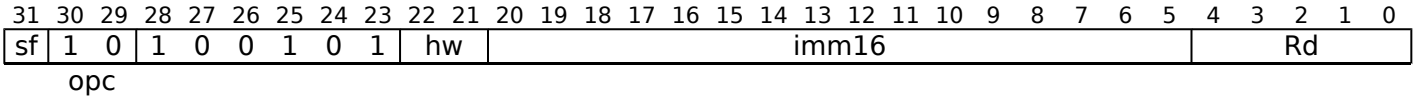
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MOV (wide immediate)

Move (wide immediate) moves a 16-bit immediate value to a register.

This is an alias of MOVZ. This means:

- The encodings in this description are named to match the encodings of MOVZ.
- The description of MOVZ gives the operational pseudocode for this instruction.



32-bit (sf == 0 && hw == 0x)

```
MOV <Wd>, #<imm>
```

is equivalent to

```
MOVZ <Wd>, #<imm16>, LSL #<shift>
```

and is the preferred disassembly when `! (IsZero(imm16) && hw != '00')`.

64-bit (sf == 1)

```
MOV <Xd>, #<imm>
```

is equivalent to

```
MOVZ <Xd>, #<imm16>, LSL #<shift>
```

and is the preferred disassembly when `! (IsZero(imm16) && hw != '00')`.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<imm>	For the 32-bit variant: is a 32-bit immediate which can be encoded in "imm16:hw". For the 64-bit variant: is a 64-bit immediate which can be encoded in "imm16:hw".
<shift>	For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16. For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Operation

The description of MOVZ gives the operational pseudocode for this instruction.

Operational information

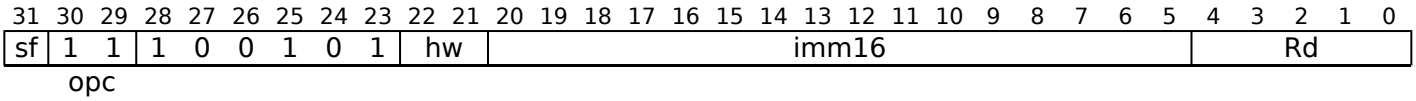
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MOVK

Move wide with keep moves an optionally-shifted 16-bit immediate value into a register, keeping other bits unchanged.



32-bit (sf == 0 && hw == 0x)

```
MOVK <Wd>, #<imm>{, LSL #<shift>}
```

64-bit (sf == 1)

```
MOVK <Xd>, #<imm>{, LSL #<shift>}
```

```
integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
integer pos;

if sf == '0' && hw<1> == '1' then UNDEFINED;
pos = UInt(hw:'0000');
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<imm>	Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.
<shift>	For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16. For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Operation

```
bits(datasize) result;

result = X[d, datasize];
result<pos+15:pos> = imm16;
X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

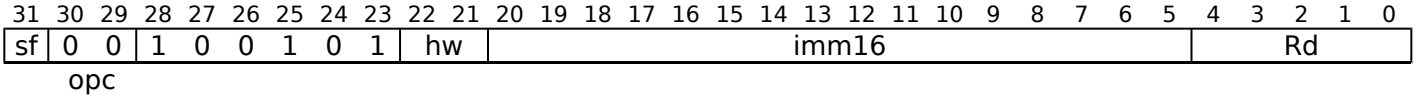
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MOVN

Move wide with NOT moves the inverse of an optionally-shifted 16-bit immediate value to a register.

This instruction is used by the alias MOV (inverted wide immediate).



32-bit (sf == 0 && hw == 0x)

```
MOVN <Wd>, #<imm>{, LSL #<shift>}
```

64-bit (sf == 1)

```
MOVN <Xd>, #<imm>{, LSL #<shift>}
```

```
integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
integer pos;

if sf == '0' && hw<1> == '1' then UNDEFINED;
pos = UInt(hw:'0000');
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<imm>	Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.
<shift>	For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16. For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Alias Conditions

Alias	Of variant	Is preferred when
MOV (inverted wide immediate)	64-bit	`! (IsZero(imm16) && hw != '00')`
MOV (inverted wide immediate)	32-bit	`! (IsZero(imm16) && hw != '00') && ! IsOnes(imm16)`

Operation

```
bits(datasize) result;

result = Zeros(datasize);

result<pos+15:pos> = imm16;
result = NOT(result);
X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

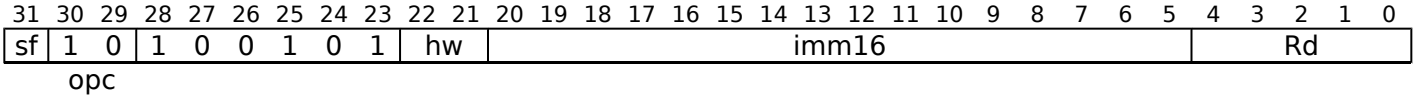
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MOVZ

Move wide with zero moves an optionally-shifted 16-bit immediate value to a register.
This instruction is used by the alias MOV (wide immediate).



32-bit (sf == 0 && hw == 0x)

```
MOVZ <Wd>, #<imm>{, LSL #<shift>}
```

64-bit (sf == 1)

```
MOVZ <Xd>, #<imm>{, LSL #<shift>}
```

```
integer d = UInt(Rd);
integer datasize = if sf == '1' then 64 else 32;
integer pos;

if sf == '0' && hw<1> == '1' then UNDEFINED;
pos = UInt(hw:'0000');
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<imm>	Is the 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.
<shift>	For the 32-bit variant: is the amount by which to shift the immediate left, either 0 (the default) or 16, encoded in the "hw" field as <shift>/16. For the 64-bit variant: is the amount by which to shift the immediate left, either 0 (the default), 16, 32 or 48, encoded in the "hw" field as <shift>/16.

Alias Conditions

Alias	Is preferred when
MOV (wide immediate)	`! (IsZero(imm16) && hw != '00')`

Operation

```
bits(datasize) result;

result = Zeros(datasize);

result<pos+15:pos> = imm16;
X[d, datasize] = result;
```

Operational information

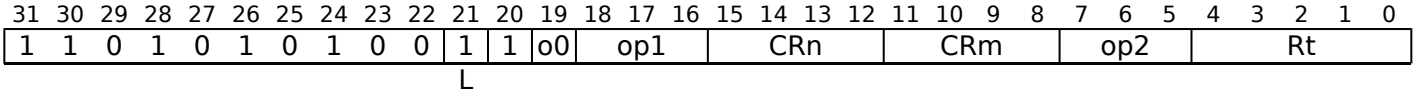
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MRS

Move System Register allows the PE to read an AArch64 System register into a general-purpose register.



MRS <Xt>, (<systemreg>|S<op0>_<op1>_<Cn>_<Cm>_<op2>)

```
AArch64.CheckSystemAccess('1':o0, op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);

integer sys_op0 = 2 + UInt(o0);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
```

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.

<systemreg> Is a System register name, encoded in the "o0:op1:CRn:CRm:op2".
The System register names are defined in *'AArch64 System Registers' in the System Register XML*.

<op0> Is an unsigned immediate, encoded in “o0”:

o0	<op0>
0	2
1	3

<op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

<Cn> Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.

<Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

<op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

Operation

```
AArch64.SysRegRead(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, t);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

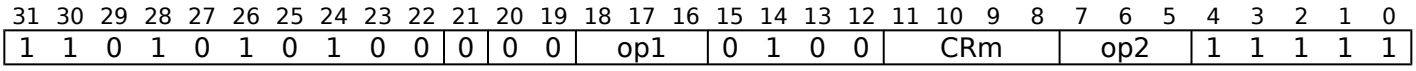
MSR (immediate)

Move immediate value to Special Register moves an immediate value to selected bits of the PSTATE. For more information, see *Process state, PSTATE*.

The bits that can be written by this instruction are:

- PSTATE.D, PSTATE.A, PSTATE.I, PSTATE.F, and PSTATE.SP.
- If *FEAT_SSBS* is implemented, PSTATE.SSBS.
- If *FEAT_PAN* is implemented, PSTATE.PAN.
- If *FEAT_UAO* is implemented, PSTATE.UAO.
- If *FEAT_DIT* is implemented, PSTATE.DIT.
- If *FEAT_MTE* is implemented, PSTATE.TCO.
- If *FEAT_NMI* is implemented, PSTATE.ALLINT.
- If FEAT_SME is implemented, PSTATE.SM and PSTATE.ZA.

This instruction is used by the aliases SMSTART, and SMSTOP.



MSR <pstatefield>, #<imm>

```
if op1 == '000' && op2 == '000' then SEE "CFINV";
if op1 == '000' && op2 == '001' then SEE "XAFLAG";
if op1 == '000' && op2 == '010' then SEE "AXFLAG";

AArch64.CheckSystemAccess('00', op1, '0100', CRm, op2, '11111', '0');
bits(2) min_EL;
boolean need_secure = FALSE;

case op1 of
    when '00x'
        min_EL = EL1;
    when '010'
        min_EL = EL1;
    when '011'
        min_EL = EL0;
    when '100'
        min_EL = EL2;
    when '101'
        if !HaveVirtHostExt() then
            UNDEFINED;
        min_EL = EL2;
    when '110'
        min_EL = EL3;
    when '111'
        min_EL = EL1;
        need_secure = TRUE;

if (UInt(PSTATE.EL) < UInt(min_EL) || (need_secure && CurrentSecurityState() != SS_Secure)) then
    UNDEFINED;

PSTATEField field;
case op1:op2 of
    when '000 011'
        if !HaveUAOExt() then UNDEFINED;
        field = PSTATEField_UAO;
    when '000 100'
        if !HavePANExt() then UNDEFINED;
        field = PSTATEField_PAN;
    when '000 101' field = PSTATEField_SP;
    when '001 000'
        if !HaveFeatNMI() then UNDEFINED;
        if CRm<3:1> != '000' then UNDEFINED;
        field = PSTATEField_ALLINT;
    when '011 010'
        if !HaveDITExt() then UNDEFINED;
        field = PSTATEField_DIT;
    when '011 011'
        case CRm of
            when '001x'
                if !HaveSME() then UNDEFINED;
                field = PSTATEField_SVCRSM;
            when '010x'
                if !HaveSME() then UNDEFINED;
                field = PSTATEField_SVCRZA;
            when '011x'
                if !HaveSME() then UNDEFINED;
                field = PSTATEField_SVCRSMZA;
            otherwise
                UNDEFINED;
    when '011 100'
        if !HaveMTEExt() then UNDEFINED;
        field = PSTATEField_TCO;
    when '011 110' field = PSTATEField_DAIFSet;
    when '011 111' field = PSTATEField_DAIFClr;
    when '011 001'
        if !HaveSSBSExt() then UNDEFINED;
        field = PSTATEField_SSBS;
    otherwise UNDEFINED;

// Check that an AArch64 MSR/MRS access to the DAIF flags is permitted
```

```
if PSTATE.EL == EL0 && field IN {PSTATEField_DAIFSet, PSTATEField_DAIFClr} then
    if !ELUsingAArch32(EL1) && ((EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') || SCTLR_EL1.UMA == '0') then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
```

Assembler Symbols

<pstatefield> Is a PSTATE field name. For the MSR instruction, this is encoded in “op1:op2:CRm”:

op1	op2	CRm	<pstatefield>	Architectural Feature
000	00x	xxxx	SEE_PSTATE	-
000	010	xxxx	SEE_PSTATE	-
000	011	xxxx	UAO	FEAT_UAO
000	100	xxxx	PAN	FEAT_PAN
000	101	xxxx	SPSel	-
000	11x	xxxx	RESERVED	-
001	000	000x	ALLINT	FEAT_NMI
001	000	001x	RESERVED	-
001	000	01xx	RESERVED	-
001	000	1xxx	RESERVED	-
001	001	xxxx	RESERVED	-
001	01x	xxxx	RESERVED	-
001	1xx	xxxx	RESERVED	-
010	xxx	xxxx	RESERVED	-
011	000	xxxx	RESERVED	-
011	001	xxxx	SSBS	FEAT_SSBS
011	010	xxxx	DIT	FEAT_DIT
011	011	000x	RESERVED	-
011	011	001x	SVCRSM	FEAT_SME
011	011	010x	SVCRZA	FEAT_SME
011	011	011x	SVCRSMZA	FEAT_SME
011	011	1xxx	RESERVED	-
011	100	xxxx	TCO	FEAT_MTE
011	101	xxxx	RESERVED	-
011	110	xxxx	DAIFSet	-
011	111	xxxx	DAIFClr	-
1xx	xxx	xxxx	RESERVED	-

For the SMSTART and SMSTOP aliases, this is encoded in "CRm<2:1>", where 0b01 specifies SVCRSM, 0b10 specifies SVCRZA, and 0b11 specifies SVCRSMZA.

<imm> Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "CRm" field. Restricted to the range 0 to 1, encoded in "CRm<0>", when <pstatefield> is ALLINT, SVCRSM, SVCRSMZA, or SVCRZA.

Alias Conditions

Alias	Is preferred when
SMSTART	`op1 == '011' && CRm == '0xx1' && op2 == '011'`
SMSTOP	`op1 == '011' && CRm == '0xx0' && op2 == '011'`

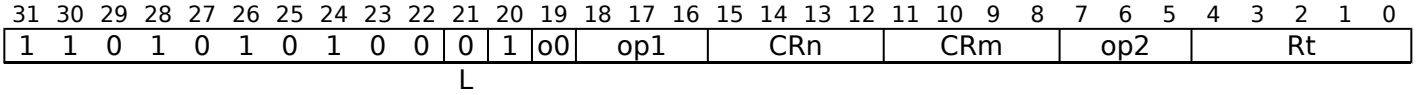
Operation

```
case field of
    when PSTATEField_SSBS
        PSTATE.SSBS = CRm<0>;
    when PSTATEField_SP
        PSTATE.SP = CRm<0>;
    when PSTATEField_DAIFSet
        PSTATE.D = PSTATE.D OR CRm<3>;
        PSTATE.A = PSTATE.A OR CRm<2>;
        PSTATE.I = PSTATE.I OR CRm<1>;
        PSTATE.F = PSTATE.F OR CRm<0>;
    when PSTATEField_DAIFClr
        PSTATE.D = PSTATE.D AND NOT(CRm<3>);
        PSTATE.A = PSTATE.A AND NOT(CRm<2>);
        PSTATE.I = PSTATE.I AND NOT(CRm<1>);
        PSTATE.F = PSTATE.F AND NOT(CRm<0>);
    when PSTATEField_PAN
        PSTATE.PAN = CRm<0>;
    when PSTATEField_UAO
        PSTATE.UAO = CRm<0>;
    when PSTATEField_DIT
        PSTATE.DIT = CRm<0>;
    when PSTATEField_TCO
        PSTATE.TCO = CRm<0>;
    when PSTATEField_ALLINT
        if (PSTATE.EL == EL1 && IsHCRXEL2Enabled() && HCRX_EL2.TALLINT == '1' && CRm<0> == '1') then
            AArch64.SystemAccessTrap(EL2, 0x18);
        PSTATE.ALLINT = CRm<0>;
    when PSTATEField_SVCRSM
        CheckSMEAccess();
        SetPSTATE_SM(CRm<0>);
    when PSTATEField_SVCRZA
        CheckSMEAccess();
        SetPSTATE_ZA(CRm<0>);
    when PSTATEField_SVCRSMZA
        CheckSMEAccess();
        SetPSTATE_SM(CRm<0>);
        SetPSTATE_ZA(CRm<0>);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MSR (register)

Move general-purpose register to System Register allows the PE to write an AArch64 System register from a general-purpose register.



MSR (<systemreg>|S<op0>_<op1>_<Cn>_<Cm>_<op2>), <Xt>

```
AArch64.CheckSystemAccess('1':o0, op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);

integer sys_op0 = 2 + UInt(o0);
integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
```

Assembler Symbols

<systemreg> Is a System register name, encoded in the "o0:op1:CRn:CRm:op2".
The System register names are defined in *'AArch64 System Registers' in the System Register XML*.

<op0> Is an unsigned immediate, encoded in “o0”:

o0	<op0>
0	2
1	3

<op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

<Cn> Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.

<Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

<op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rt" field.

Operation

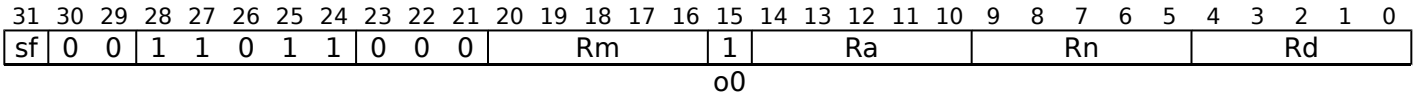
```
AArch64.SysRegWrite(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, t);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MSUB

Multiply-Subtract multiplies two register values, subtracts the product from a third register value, and writes the result to the destination register.

This instruction is used by the alias MNEG.



32-bit (sf == 0)

```
MSUB <Wd>, <Wn>, <Wm>, <Wa>
```

64-bit (sf == 1)

```
MSUB <Xd>, <Xn>, <Xm>, <Xa>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
integer destsize = if sf == '1' then 64 else 32;
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Wa>	Is the 32-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Xa>	Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

Alias Conditions

Alias	Is preferred when
MNEG	`Ra == '11111'`

Operation

```
bits(destsize) operand1 = X[n, destsize];
bits(destsize) operand2 = X[m, destsize];
bits(destsize) operand3 = X[a, destsize];

integer result;

result = UInt(operand3) - (UInt(operand1) * UInt(operand2));
X[d, destsize] = result<destsize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

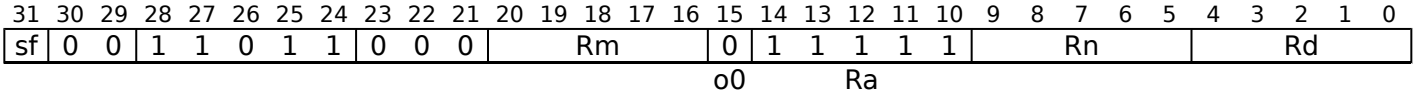
MUL

Multiply

: `Rd = Rn * Rm`.

This is an alias of MADD. This means:

- The encodings in this description are named to match the encodings of MADD.
- The description of MADD gives the operational pseudocode for this instruction.



32-bit (sf == 0)

`MUL <Wd>, <Wn>, <Wm>`

is equivalent to

`MADD <Wd>, <Wn>, <Wm>, WZR`

and is always the preferred disassembly.

64-bit (sf == 1)

`MUL <Xd>, <Xn>, <Xm>`

is equivalent to

`MADD <Xd>, <Xn>, <Xm>, XZR`

and is always the preferred disassembly.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of MADD gives the operational pseudocode for this instruction.

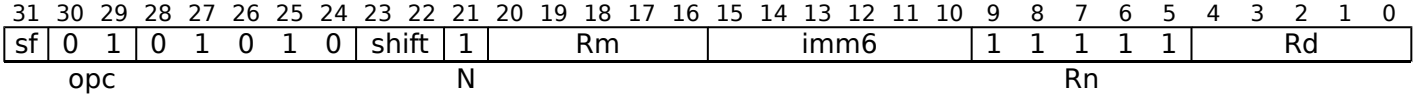
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MVN

Bitwise NOT writes the bitwise inverse of a register value to the destination register.

This is an alias of ORN (shifted register). This means:

- The encodings in this description are named to match the encodings of ORN (shifted register).
- The description of ORN (shifted register) gives the operational pseudocode for this instruction.



32-bit (sf == 0)

```
MVN <Wd>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
ORN <Wd>, WZR, <Wm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

64-bit (sf == 1)

```
MVN <Xd>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
ORN <Xd>, XZR, <Xm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wm> Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xm> Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

Operation

The description of ORN (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

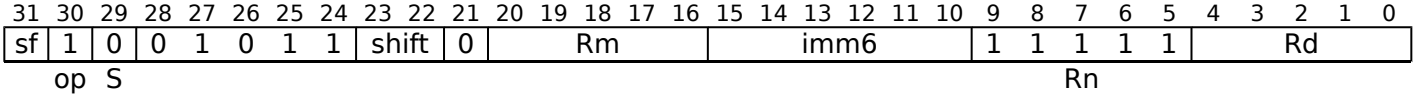
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

NEG (shifted register)

Negate (shifted register) negates an optionally-shifted register value, and writes the result to the destination register.

This is an alias of SUB (shifted register). This means:

- The encodings in this description are named to match the encodings of SUB (shifted register).
- The description of SUB (shifted register) gives the operational pseudocode for this instruction.



32-bit (sf == 0)

```
NEG <Wd>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
SUB <Wd>, WZR, <Wm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

64-bit (sf == 1)

```
NEG <Xd>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
SUB <Xd>, XZR, <Xm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wm> Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xm> Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Operation

The description of SUB (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

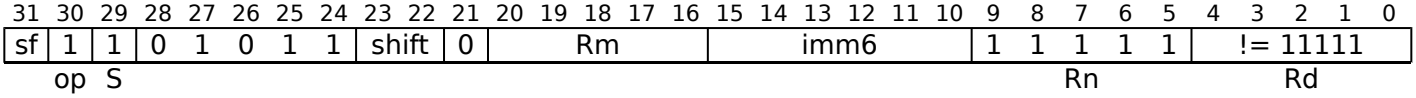
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

NEGS

Negate, setting flags, negates an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This is an alias of SUBS (shifted register). This means:

- The encodings in this description are named to match the encodings of SUBS (shifted register).
- The description of SUBS (shifted register) gives the operational pseudocode for this instruction.



32-bit (sf == 0)

```
NEGS <Wd>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
SUBS <Wd>, WZR, <Wm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

64-bit (sf == 1)

```
NEGS <Xd>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
SUBS <Xd>, XZR, <Xm> {, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wm> Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xm> Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in "shift":

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Operation

The description of SUBS (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

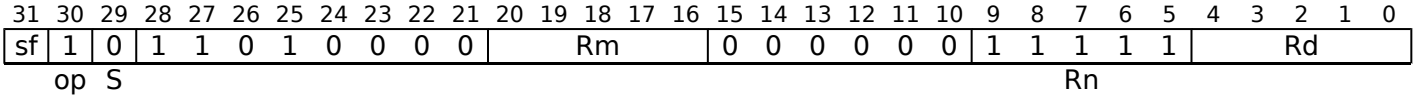
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

NGC

Negate with Carry negates the sum of a register value and the value of NOT (Carry flag), and writes the result to the destination register.

This is an alias of SBC. This means:

- The encodings in this description are named to match the encodings of SBC.
- The description of SBC gives the operational pseudocode for this instruction.



32-bit (sf == 0)

```
NGC <Wd>, <Wm>
```

is equivalent to

```
SBC <Wd>, WZR, <Wm>
```

and is always the preferred disassembly.

64-bit (sf == 1)

```
NGC <Xd>, <Xm>
```

is equivalent to

```
SBC <Xd>, XZR, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wm>	Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xm>	Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

Operation

The description of SBC gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

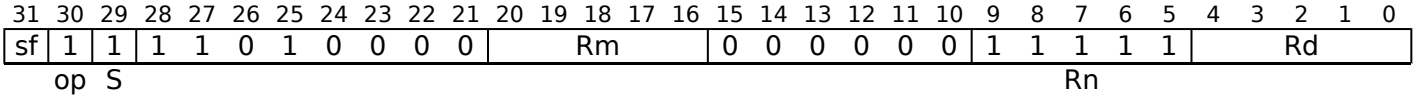
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

NGCS

Negate with Carry, setting flags, negates the sum of a register value and the value of NOT (Carry flag), and writes the result to the destination register. It updates the condition flags based on the result.

This is an alias of SBCS. This means:

- The encodings in this description are named to match the encodings of SBCS.
- The description of SBCS gives the operational pseudocode for this instruction.



32-bit (sf == 0)

```
NGCS <Wd>, <Wm>
```

is equivalent to

```
SBCS <Wd>, WZR, <Wm>
```

and is always the preferred disassembly.

64-bit (sf == 1)

```
NGCS <Xd>, <Xm>
```

is equivalent to

```
SBCS <Xd>, XZR, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wm>	Is the 32-bit name of the general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xm>	Is the 64-bit name of the general-purpose source register, encoded in the "Rm" field.

Operation

The description of SBCS gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

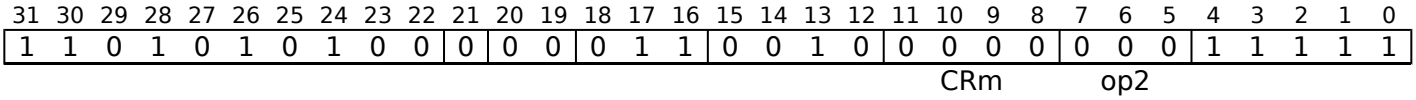
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

NOP

No Operation does nothing, other than advance the value of the program counter by 4. This instruction can be used for instruction alignment purposes.

Note

The timing effects of including a `NOP` instruction in a program are not guaranteed. It can increase execution time, leave it unchanged, or even reduce it. Therefore, `NOP` instructions are not suitable for timing loops.



`NOP`

```
// Empty.
```

Operation

```
// do nothing
```

Operational information

If PSTATE.DIT is 1:

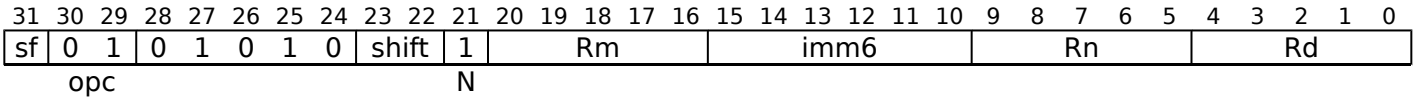
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ORN (shifted register)

Bitwise OR NOT (shifted register) performs a bitwise (inclusive) OR of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register.

This instruction is used by the alias MVN.



32-bit (sf == 0)

```
ORN <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit (sf == 1)

```
ORN <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

Alias Conditions

Alias	Is preferred when
MVN	`Rn == '11111'`

Operation

```
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);
bits(datasize) result;

operand2 = NOT(operand2);

result = operand1 OR operand2;
X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

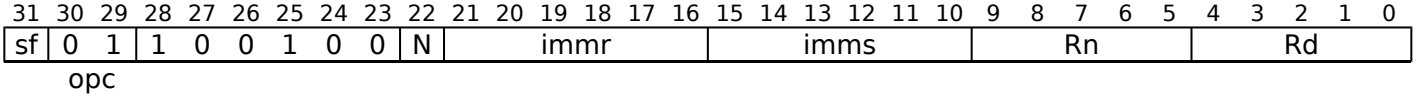
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ORR (immediate)

Bitwise OR (immediate) performs a bitwise (inclusive) OR of a register value and an immediate register value, and writes the result to the destination register.

This instruction is used by the alias MOV (bitmask immediate).



32-bit (sf == 0 && N == 0)

```
ORR <Wd|WSP>, <Wn>, #<imm>
```

64-bit (sf == 1)

```
ORR <Xd|SP>, <Xn>, #<imm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;
if sf == '0' && N != '0' then UNDEFINED;
(imm, -) = DecodeBitMasks(N, imms, immr, TRUE, datasize);
```

Assembler Symbols

<Wd|WSP>	Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd|SP>	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<imm>	For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr". For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

Alias Conditions

Alias	Is preferred when
MOV (bitmask immediate)	`Rn == '11111' && ! MoveWidePreferred(sf, N, imms, immr)`

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n, datasize];

result = operand1 OR imm;
if d == 31 then
    SP[] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

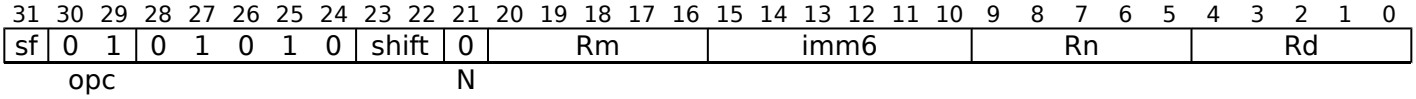
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ORR (shifted register)

Bitwise OR (shifted register) performs a bitwise (inclusive) OR of a register value and an optionally-shifted register value, and writes the result to the destination register.

This instruction is used by the alias MOV (register).



32-bit (sf == 0)

```
ORR <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit (sf == 1)

```
ORR <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

Alias Conditions

Alias	Is preferred when
MOV (register)	`shift == '00' && imm6 == '000000' && Rn == '11111'`

Operation

```
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);
bits(datasize) result;

result = operand1 OR operand2;
X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PACDA, PACDZA

Pointer Authentication Code for Data address, using key A. This instruction computes and inserts a pointer authentication code for a data address, using a modifier and key A.

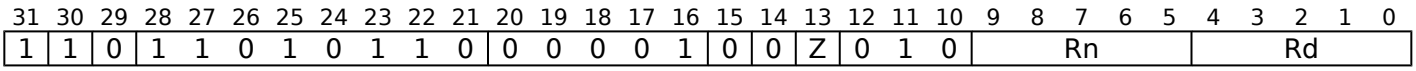
The address is in the general-purpose register that is specified by <Xd>.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACDA.
- The value zero, for PACDZA.

Integer

(FEAT_PAuth)



PACDA (Z == 0)

```
PACDA <Xd>, <Xn|SP>
```

PACDZA (Z == 1 && Rn == 11111)

```
PACDZA <Xd>
```

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;

if Z == '0' then // PACDA
    if n == 31 then source_is_sp = TRUE;
else // PACDZA
    if n != 31 then UNDEFINED;
```

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn|SP>	Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

```
if source_is_sp then
    X[d, 64] = AddPACDA(X[d, 64], SP[]);
else
    X[d, 64] = AddPACDA(X[d, 64], X[n, 64]);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PACDB, PACDZB

Pointer Authentication Code for Data address, using key B. This instruction computes and inserts a pointer authentication code for a data address, using a modifier and key B.

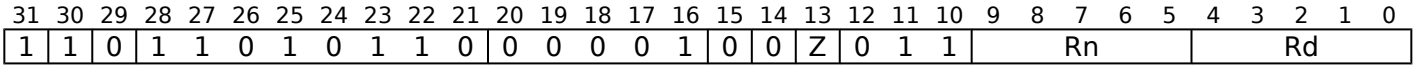
The address is in the general-purpose register that is specified by <Xd>.

The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACDB.
- The value zero, for PACDZB.

Integer

(FEAT_PAuth)



PACDB (Z == 0)

```
PACDB <Xd>, <Xn|SP>
```

PACDZB (Z == 1 && Rn == 11111)

```
PACDZB <Xd>
```

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;

if Z == '0' then // PACDB
    if n == 31 then source_is_sp = TRUE;
else // PACDZB
    if n != 31 then UNDEFINED;
```

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn|SP>	Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

```
if source_is_sp then
    X[d, 64] = AddPACDB(X[d, 64], SP[]);
else
    X[d, 64] = AddPACDB(X[d, 64], X[n, 64]);
```

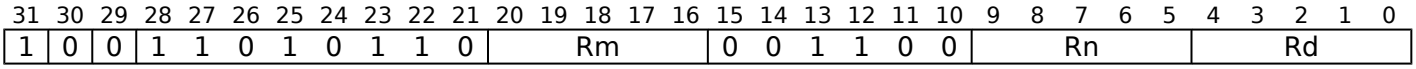
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PACGA

Pointer Authentication Code, using Generic key. This instruction computes the pointer authentication code for an address in the first source register, using a modifier in the second source register, and the Generic key. The computed pointer authentication code is returned in the upper 32 bits of the destination register.

Integer

(FEAT_PAuth)



PACGA <Xd>, <Xn>, <Xm|SP>

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if !HavePACExt() then
    UNDEFINED;

if m == 31 then source_is_sp = TRUE;
```

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm|SP>	Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Rm" field.

Operation

```
if source_is_sp then
    X[d, 64] = AddPACGA(X[n, 64], SP[]);
else
    X[d, 64] = AddPACGA(X[n, 64], X[m, 64]);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA

Pointer Authentication Code for Instruction address, using key A. This instruction computes and inserts a pointer authentication code for an instruction address, using a modifier and key A.

The address is:

- In the general-purpose register that is specified by <Xd> for PACIA and PACIZA.
- In X17, for PACIA1716.
- In X30, for PACIASP and PACIAZ.

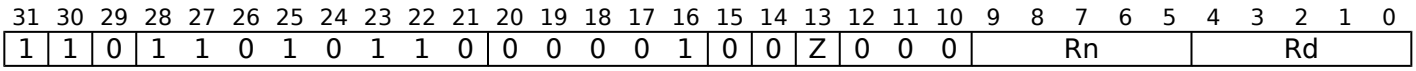
The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACIA.
- The value zero, for PACIZA and PACIAZ.
- In X16, for PACIA1716.
- In SP, for PACIASP.

It has encodings from 2 classes: Integer and System

Integer

(FEAT_PAuth)



PACIA (Z == 0)

```
PACIA <Xd>, <Xn|SP>
```

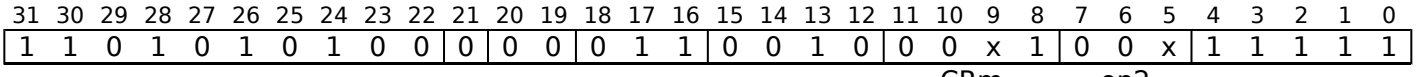
PACIZA (Z == 1 && Rn == 11111)

```
PACIZA <Xd>
```

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;

if Z == '0' then // PACIA
    if n == 31 then source_is_sp = TRUE;
else // PACIZA
    if n != 31 then UNDEFINED;
```

System

(FEAT_PAuth)



PACIA1716 (CRm == 0001 && op2 == 000)

PACIA1716

PACIASP (CRm == 0011 && op2 == 001)

PACIASP

PACIAZ (CRm == 0011 && op2 == 000)

PACIAZ

```
integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
    when '0011 000'    // PACIAZ
        d = 30;
        n = 31;
    when '0011 001'    // PACIASP
        d = 30;
        source_is_sp = TRUE;
        if HaveBTIExt() then
            // Check for branch target compatibility between PSTATE.BTYPE
            // and implicit branch target of PACIASP instruction.
            SetBTypeCompatible(BTypeCompatible_PACIXSP());

    when '0001 000'    // PACIA1716
        d = 17;
        n = 16;
    when '0001 010' SEE "PACIB";
    when '0001 100' SEE "AUTIA";
    when '0001 110' SEE "AUTIB";
    when '0011 01x' SEE "PACIB";
    when '0011 10x' SEE "AUTIA";
    when '0011 11x' SEE "AUTIB";
    when '0000 111' SEE "XPACLRI";
    otherwise SEE "HINT";
```

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn|SP>	Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

```
if HavePACExt() then
    if source_is_sp then
        X[d, 64] = AddPACIA(X[d, 64], SP[]);
    else
        X[d, 64] = AddPACIA(X[d, 64], X[n, 64]);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB

Pointer Authentication Code for Instruction address, using key B. This instruction computes and inserts a pointer authentication code for an instruction address, using a modifier and key B.

The address is:

- In the general-purpose register that is specified by <Xd> for PACIB and PACIZB.
- In X17, for PACIB1716.
- In X30, for PACIBSP and PACIBZ.

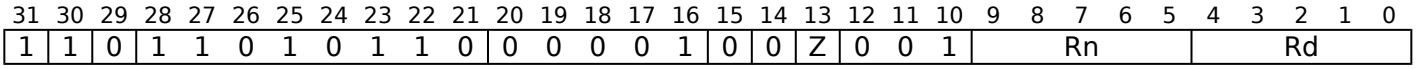
The modifier is:

- In the general-purpose register or stack pointer that is specified by <Xn|SP> for PACIB.
- The value zero, for PACIZB and PACIBZ.
- In X16, for PACIB1716.
- In SP, for PACIBSP.

It has encodings from 2 classes: Integer and System

Integer

(FEAT_PAuth)



PACIB (Z == 0)

```
PACIB <Xd>, <Xn|SP>
```

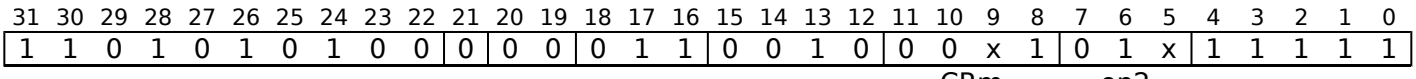
PACIZB (Z == 1 && Rn == 11111)

```
PACIZB <Xd>
```

```
boolean source_is_sp = FALSE;
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HavePACExt() then
    UNDEFINED;

if Z == '0' then // PACIB
    if n == 31 then source_is_sp = TRUE;
else // PACIZB
    if n != 31 then UNDEFINED;
```

System

(FEAT_PAuth)



PACIB1716 (CRm == 0001 && op2 == 010)

PACIB1716

PACIBSP (CRm == 0011 && op2 == 011)

PACIBSP

PACIBZ (CRm == 0011 && op2 == 010)

PACIBZ

```
integer d;
integer n;
boolean source_is_sp = FALSE;

case CRm:op2 of
    when '0011 010'    // PACIBZ
        d = 30;
        n = 31;
    when '0011 011'    // PACIBSP
        d = 30;
        source_is_sp = TRUE;
        if HaveBTIExt() then
            // Check for branch target compatibility between PSTATE.BTYPE
            // and implicit branch target of PACIBSP instruction.
            SetBTypeCompatible(BTypeCompatible_PACIXSP());
    when '0001 010'    // PACIB1716
        d = 17;
        n = 16;
    when '0001 000' SEE "PACIA";
    when '0001 100' SEE "AUTIA";
    when '0001 110' SEE "AUTIB";
    when '0011 00x' SEE "PACIA";
    when '0011 10x' SEE "AUTIA";
    when '0011 11x' SEE "AUTIB";
    when '0000 111' SEE "XPACLRI";
    otherwise SEE "HINT";
```

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn|SP>	Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Rn" field.

Operation

```
if HavePACExt() then
    if source_is_sp then
        X[d, 64] = AddPACIB(X[d, 64], SP[]);
    else
        X[d, 64] = AddPACIB(X[d, 64], X[n, 64]);
```

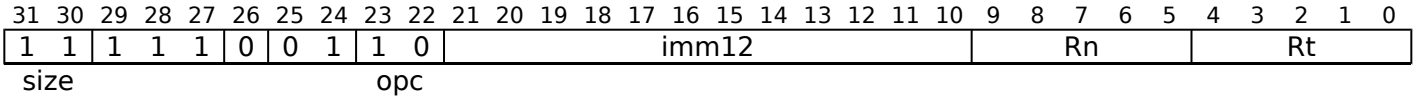
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PRFM (immediate)

Prefetch Memory (immediate) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of a PRFM instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory*.

For information about memory accesses, see *Load/Store addressing modes*.




```
PRFM (<prfop>|#<imm5>), [<Xn|SP>{, #<pimm>}]
```

```
bits(64) offset = LSL(ZeroExtend(imm12, 64), 3);
```

Assembler Symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>.

<type> is one of:

PLD
: Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.

PLI
: Preload instructions, encoded in the "Rt<4:3>" field as 0b01.

PST
: Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.

<target> is one of:

L1
: Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.

L2
: Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.

L3
: Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.

<policy> is one of:

KEEP
: Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.

STRM
: Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see *Prefetch memory*.

For other encodings of the "Rt" field, use <imm5>.

<imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field.

This syntax is only for encodings that are not accessible using <prfop>.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<pimm> Is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
```

Operation

```
bits(64) address;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);

if n == 31 then
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

Prefetch(address, t<4:0>);
```

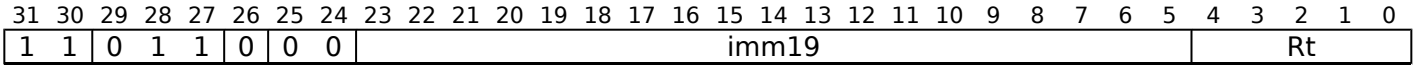
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PRFM (literal)

Prefetch Memory (literal) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of a PRFM instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory*.

For information about memory accesses, see *Load/Store addressing modes*.




```
PRFM (<prfop>|#<imm5>), <label>
```

```
integer t = UInt(Rt);
bits(64) offset;

offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>.

<type> is one of:

PLD
: Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.

PLI
: Preload instructions, encoded in the "Rt<4:3>" field as 0b01.

PST
: Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.

<target> is one of:

L1
: Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.

L2
: Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.

L3
: Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.

<policy> is one of:

KEEP
: Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.

STRM
: Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see *Prefetch memory*.

For other encodings of the "Rt" field, use <imm5>.

<imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field.

This syntax is only for encodings that are not accessible using <prfop>.

<label> Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```
bits(64) address = PC[] + offset;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);

Prefetch(address, t<4:0>);
```

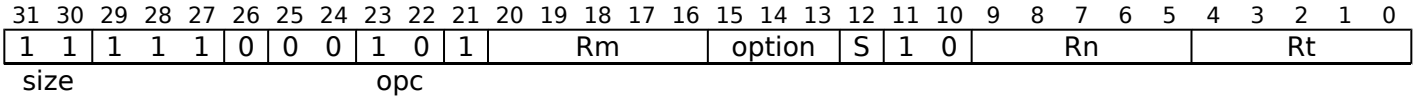
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PRFM (register)

Prefetch Memory (register) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of a PRFM instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory*.

For information about memory accesses, see *Load/Store addressing modes*.




```
PRFM (<prfop>|#<imm5>), [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 3 else 0;
```

Assembler Symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>.

<type> is one of:

PLD
Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.

PLI
Preload instructions, encoded in the "Rt<4:3>" field as 0b01.

PST
Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.

<target> is one of:

L1
Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.

L2
Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.

L3
Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.

<policy> is one of:

KEEP
Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.

STRM
Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see *Prefetch memory*.

For other encodings of the "Rt" field, use <imm5>.

<imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field. This syntax is only for encodings that are not accessible using <prfop>.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in "S":

S	<amount>
0	#0
1	#3

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);

if n == 31 then
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

Prefetch(address, t<4:0>);
```

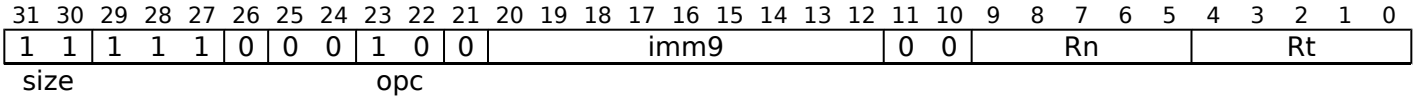
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PRFUM

Prefetch Memory (unscaled offset) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of a PRFUM instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory*.

For information about memory accesses, see *Load/Store addressing modes*.




```
PRFUM (<prfop>|#<imm5>), [<Xn|SP>{, #<simm>}]
```

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>.

<type> is one of:

PLD
: Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.

PLI
: Preload instructions, encoded in the "Rt<4:3>" field as 0b01.

PST
: Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.

<target> is one of:

L1
: Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.

L2
: Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.

L3
: Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.

<policy> is one of:

KEEP
: Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.

STRM
: Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see *Prefetch memory*.

For other encodings of the "Rt" field, use <imm5>.

<imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field.

This syntax is only for encodings that are not accessible using <prfop>.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
```

Operation

```
bits(64) address;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);

if n == 31 then
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

Prefetch(address, t<4:0>);
```

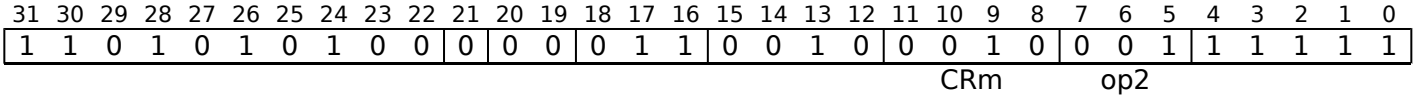
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PSB CSYNC

Profiling Synchronization Barrier. This instruction is a barrier that ensures that all existing profiling data for the current PE has been formatted, and profiling buffer addresses have been translated such that all writes to the profiling buffer have been initiated. A following DSB instruction completes when the writes to the profiling buffer have completed.

If the Statistical Profiling Extension is not implemented, this instruction executes as a NOP.

System
(FEAT_SPE)



PSB CSYNC

```
if !HaveStatisticalProfiling() then EndOfInstruction();
```

Operation

```
ProfilingSynchronizationBarrier();
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PSSBB

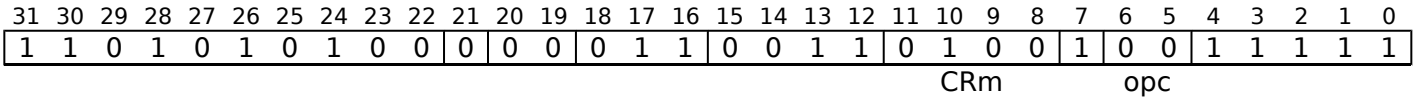
Physical Speculative Store Bypass Barrier is a memory barrier which prevents speculative loads from bypassing earlier stores to the same physical address.

The semantics of the Physical Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the PSSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order before the PSSBB.
- When a load to a location appears in program order before the PSSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order after the PSSBB.

This is an alias of DSB. This means:

- The encodings in this description are named to match the encodings of DSB.
- The description of DSB gives the operational pseudocode for this instruction.



PSSBB

is equivalent to

DSB #4

and is always the preferred disassembly.

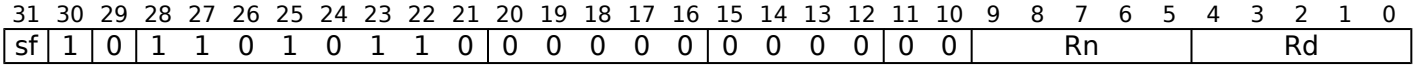
Operation

The description of DSB gives the operational pseudocode for this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

RBIT

Reverse Bits reverses the bit order in a register.



32-bit (sf == 0)

```
RBIT <Wd>, <Wn>
```

64-bit (sf == 1)

```
RBIT <Xd>, <Xn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
bits(datasize) operand = X[n, datasize];
bits(datasize) result;

for i = 0 to datasize-1
    result<(datasize-1)-i> = operand<i>;

X[d, datasize] = result;
```

Operational information

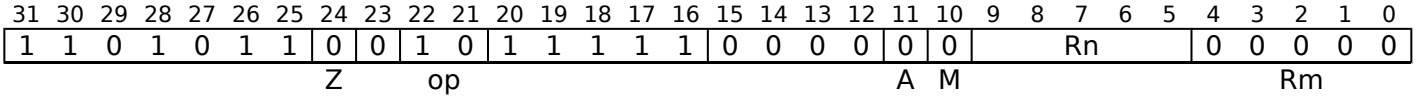
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

RET

Return from subroutine branches unconditionally to an address in a register, with a hint that this is a subroutine return.



RET {<Xn>}

```
integer n = UInt(Rn);
```

Assembler Symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field. Defaults to X30 if absent.

Operation

```
bits(64) target = X[n, 64];

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '00';

BranchTo(target, BranchType_RET, FALSE);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

RETAA, RETAB

Return from subroutine, with pointer authentication. This instruction authenticates the address that is held in LR, using SP as the modifier and the specified key, branches to the authenticated address, with a hint that this instruction is a subroutine return.

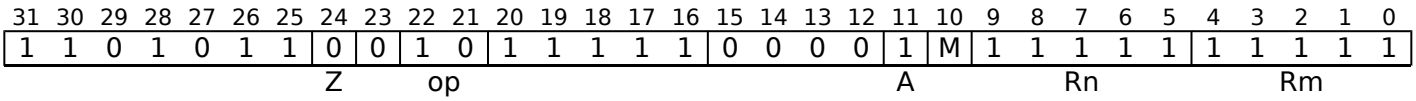
Key A is used for RETAA, and key B is used for RETAB.

If the authentication passes, the PE continues execution at the target of the branch. If the authentication fails, a Translation fault is generated.

The authenticated address is not written back to LR.

Integer

(FEAT_PAuth)



RETAA (M == 0)

```
RETAA
```

RETAB (M == 1)

```
RETAB
```

```
boolean use_key_a = (M == '0');

if !HavePACExt() then
    UNDEFINED;
```

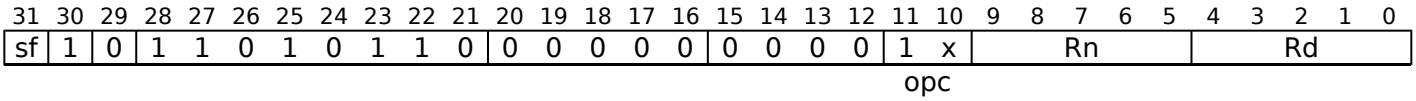
Operation

```
bits(64) target = X[30, 64];

bits(64) modifier = SP[];

if use_key_a then
    target = AuthIA(target, modifier, TRUE);
else
    target = AuthIB(target, modifier, TRUE);

// Value in BTypeNext will be used to set PSTATE.BTYPE
BTypeNext = '00';

BranchTo(target, BranchType_RET, FALSE);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

REV

Reverse Bytes reverses the byte order in a register.
This instruction is used by the pseudo-instruction REV64.



32-bit (sf == 0 && opc == 10)

```
REV <Wd>, <Wn>
```

64-bit (sf == 1 && opc == 11)

```
REV <Xd>, <Xn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;

integer container_size;
case opc of
    when '00'
        Unreachable();
    when '01'
        container_size = 16;
    when '10'
        container_size = 32;
    when '11'
        if sf == '0' then UNDEFINED;
        container_size = 64;
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
bits(datasize) operand = X[n, datasize];
bits(datasize) result;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
for c = 0 to containers-1
    rev_index = index + ((elements_per_container - 1) * 8);
    for e = 0 to elements_per_container-1
        result<rev_index+7:rev_index> = operand<index+7:index>;
        index = index + 8;
        rev_index = rev_index - 8;

X[d, datasize] = result;
```

Operational information

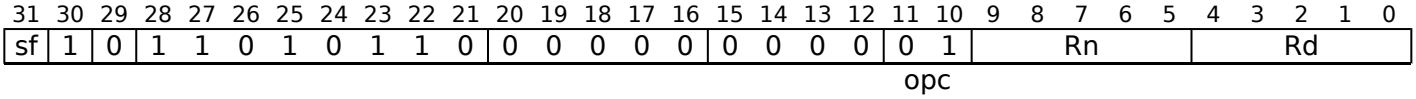
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

REV16

Reverse bytes in 16-bit halfwords reverses the byte order in each 16-bit halfword of a register.



32-bit (sf == 0)

```
REV16 <Wd>, <Wn>
```

64-bit (sf == 1)

```
REV16 <Xd>, <Xn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;

integer container_size;
case opc of
    when '00'
        Unreachable();
    when '01'
        container_size = 16;
    when '10'
        container_size = 32;
    when '11'
        if sf == '0' then UNDEFINED;
        container_size = 64;
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
bits(datasize) operand = X[n, datasize];
bits(datasize) result;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
for c = 0 to containers-1
    rev_index = index + ((elements_per_container - 1) * 8);
    for e = 0 to elements_per_container-1
        result<rev_index+7:rev_index> = operand<index+7:index>;
        index = index + 8;
        rev_index = rev_index - 8;

X[d, datasize] = result;
```

Operational information

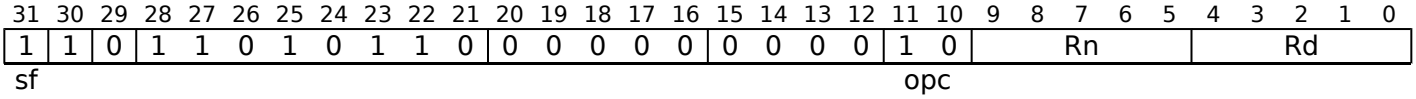
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

REV32

Reverse bytes in 32-bit words reverses the byte order in each 32-bit word of a register.



REV32 <Xd>, <Xn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer datasize = if sf == '1' then 64 else 32;

integer container_size;
case opc of
    when '00'
        Unreachable();
    when '01'
        container_size = 16;
    when '10'
        container_size = 32;
    when '11'
        if sf == '0' then UNDEFINED;
        container_size = 64;
```

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
bits(datasize) operand = X[n, datasize];
bits(datasize) result;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV 8;
integer index = 0;
integer rev_index;
for c = 0 to containers-1
    rev_index = index + ((elements_per_container - 1) * 8);
    for e = 0 to elements_per_container-1
        result<rev_index+7:rev_index> = operand<index+7:index>;
        index = index + 8;
        rev_index = rev_index - 8;

X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

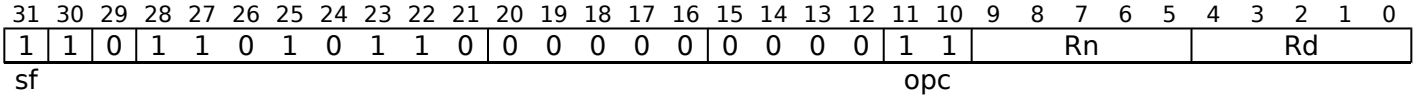
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

REV64

Reverse Bytes reverses the byte order in a 64-bit general-purpose register.
When assembling for Armv8.2, an assembler must support this pseudo-instruction. It is OPTIONAL whether an assembler supports this pseudo-instruction when assembling for an architecture earlier than Armv8.2.

This is a pseudo-instruction of REV. This means:

- The encodings in this description are named to match the encodings of REV.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of REV gives the operational pseudocode for this instruction.



64-bit

```
REV64 <Xd>, <Xn>
```

is equivalent to

```
REV <Xd>, <Xn>
```

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of REV gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

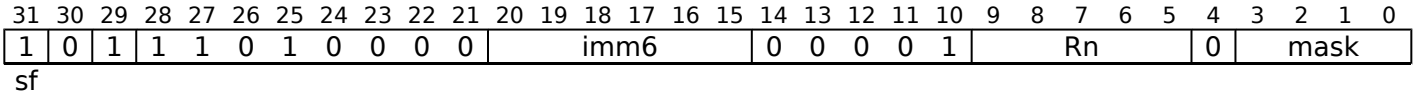
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

RMIF

Performs a rotation right of a value held in a general purpose register by an immediate value, and then inserts a selection of the bottom four bits of the result of the rotation into the PSTATE flags, under the control of a second immediate mask.

Integer
(FEAT_FlagM)




```
RMIF <Xn>, #<shift>, #<mask>
```

```
if !HaveFlagManipulateExt() then UNDEFINED;
integer lsb = UInt(imm6);
integer n = UInt(Rn);
```

Assembler Symbols

<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<shift>	Is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,
<mask>	Is the flag bit mask, an immediate in the range 0 to 15, which selects the bits that are inserted into the NZCV condition flags, encoded in the "mask" field.

Operation

```
bits(4) tmp;
bits(64) tmpreg = X[n, 64];
tmp = (tmpreg:tmpreg)<lsb+3:lsb>;
if mask<3> == '1' then PSTATE.N = tmp<3>;
if mask<2> == '1' then PSTATE.Z = tmp<2>;
if mask<1> == '1' then PSTATE.C = tmp<1>;
if mask<0> == '1' then PSTATE.V = tmp<0>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

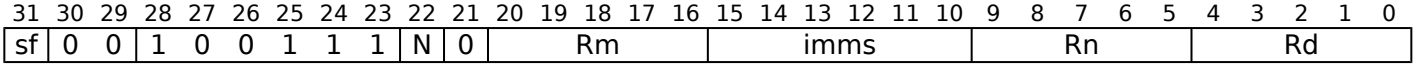
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ROR (immediate)

Rotate right (immediate) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

This is an alias of EXTR. This means:

- The encodings in this description are named to match the encodings of EXTR.
- The description of EXTR gives the operational pseudocode for this instruction.



32-bit (sf == 0 && N == 0 && imms == 0xxxxx)

```
ROR <Wd>, <Ws>, #<shift>
```

is equivalent to

```
EXTR <Wd>, <Ws>, <Ws>, #<shift>
```

and is the preferred disassembly when `Rn == Rm`.

64-bit (sf == 1 && N == 1)

```
ROR <Xd>, <Xs>, #<shift>
```

is equivalent to

```
EXTR <Xd>, <Xs>, <Xs>, #<shift>
```

and is the preferred disassembly when `Rn == Rm`.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Ws>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xs>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" and "Rm" fields.
<shift>	For the 32-bit variant: is the amount by which to rotate, in the range 0 to 31, encoded in the "imms" field. For the 64-bit variant: is the amount by which to rotate, in the range 0 to 63, encoded in the "imms" field.

Operation

The description of EXTR gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

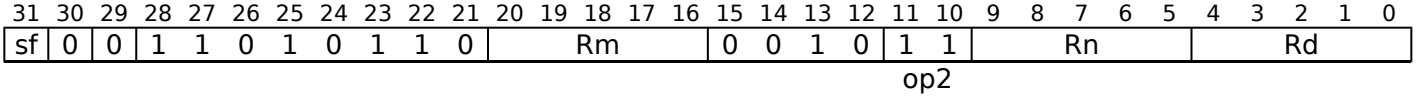
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ROR (register)

Rotate Right (register) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This is an alias of RORV. This means:

- The encodings in this description are named to match the encodings of RORV.
- The description of RORV gives the operational pseudocode for this instruction.



32-bit (sf == 0)

```
ROR <Wd>, <Wn>, <Wm>
```

is equivalent to

```
RORV <Wd>, <Wn>, <Wm>
```

and is always the preferred disassembly.

64-bit (sf == 1)

```
ROR <Xd>, <Xn>, <Xm>
```

is equivalent to

```
RORV <Xd>, <Xn>, <Xm>
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

The description of RORV gives the operational pseudocode for this instruction.

Operational information

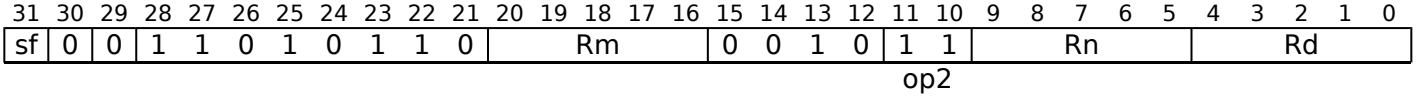
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

RORV

Rotate Right Variable provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The remainder obtained by dividing the second source register by the data size defines the number of bits by which the first source register is right-shifted.

This instruction is used by the alias ROR (register).



32-bit (sf == 0)

```
RORV <Wd>, <Wn>, <Wm>
```

64-bit (sf == 1)

```
RORV <Xd>, <Xn>, <Xm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ShiftType shift_type = DecodeShift(op2);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding a shift amount from 0 to 31 in its bottom 5 bits, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding a shift amount from 0 to 63 in its bottom 6 bits, encoded in the "Rm" field.

Operation

```
bits(datasize) result;
bits(datasize) operand2 = X[m, datasize];

result = ShiftReg(n, shift_type, UInt(operand2) MOD datasize, datasize);
X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SB

Speculation Barrier is a barrier that controls speculation.

The semantics of the Speculation Barrier are that the execution, until the barrier completes, of any instruction that appears later in the program order than the barrier:

- Cannot be performed speculatively to the extent that such speculation can be observed through side-channels as a result of control flow speculation or data value speculation.
- Can be speculatively executed as a result of predicting that a potentially exception generating instruction has not generated an exception.

In particular, any instruction that appears later in the program order than the barrier cannot cause a speculative allocation into any caching structure where the allocation of that entry could be indicative of any data value present in memory or in the registers.

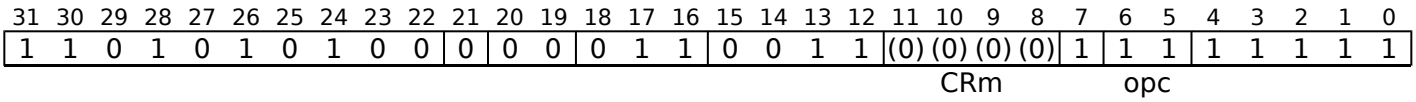
The SB instruction:

- Cannot be speculatively executed as a result of control flow speculation or data value speculation.
- Can be speculatively executed as a result of predicting that a potentially exception generating instruction has not generated an exception. The potentially exception generating instruction can complete once it is known not to be speculative, and all data values generated by instructions appearing in program order before the SB instruction have their predicted values confirmed.

When the prediction of the instruction stream is not informed by data taken from the register outputs of the speculative execution of instructions appearing in program order after an uncompleted SB instruction, the SB instruction has no effect on the use of prediction resources to predict the instruction stream that is being fetched.

System

(FEAT_SB)



SB

```
if !HaveSBExt() then UNDEFINED;
```

Operation

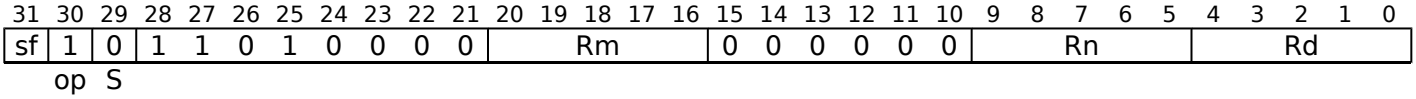
```
SpeculationBarrier();
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SBC

Subtract with Carry subtracts a register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

This instruction is used by the alias NGC.



32-bit (sf == 0)

```
SBC <Wd>, <Wn>, <Wm>
```

64-bit (sf == 1)

```
SBC <Xd>, <Xn>, <Xm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Alias Conditions

Alias	Is preferred when
NGC	`Rn == '11111'`

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = X[m, datasize];

operand2 = NOT(operand2);

(result, -) = AddWithCarry(operand1, operand2, PSTATE.C);

X[d, datasize] = result;
```

Operational information

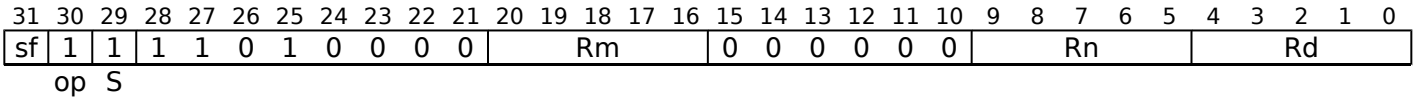
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

SBCS

Subtract with Carry, setting flags, subtracts a register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias NGCS.



32-bit (sf == 0)

```
SBCS <Wd>, <Wn>, <Wm>
```

64-bit (sf == 1)

```
SBCS <Xd>, <Xn>, <Xm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Alias Conditions

Alias	Is preferred when
NGCS	`Rn == '11111'`

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = X[m, datasize];
bits(4) nzcv;

operand2 = NOT(operand2);

(result, nzcv) = AddWithCarry(operand1, operand2, PSTATE.C);

PSTATE.<N,Z,C,V> = nzcv;

X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

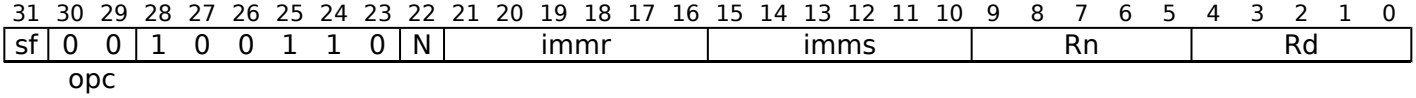
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

SBFIZ

Signed Bitfield Insert in Zeros copies a bitfield of <width> bits from the least significant bits of the source register to bit position <lsb> of the destination register, setting the destination bits below the bitfield to zero, and the bits above the bitfield to a copy of the most significant bit of the bitfield.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.



32-bit (sf == 0 && N == 0)

```
SBFIZ <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
SBFM <Wd>, <Wn>, #(-<lsb> MOD 32), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

64-bit (sf == 1 && N == 1)

```
SBFIZ <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
SBFM <Xd>, <Xn>, #(-<lsb> MOD 64), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<lsb>	For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31. For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.
<width>	For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>. For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

SBFM

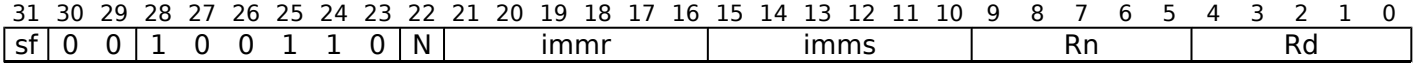
Signed Bitfield Move is usually accessed via one of its aliases, which are always preferred for disassembly.

If <imms> is greater than or equal to <immr>, this copies a bitfield of (<imms>-<immr>+1) bits starting from bit position <immr> in the source register to the least significant bits of the destination register.

If <imms> is less than <immr>, this copies a bitfield of (<imms>+1) bits from the least significant bits of the source register to bit position (regsize-<immr>) of the destination register, where regsize is the destination register size of 32 or 64 bits.

In both cases the destination bits below the bitfield are set to zero, and the bits above the bitfield are set to a copy of the most significant bit of the bitfield.

This instruction is used by the aliases ASR (immediate), SBFIZ, SBFX, SXTB, SXTH, and SXTW.



32-bit (sf == 0 && N == 0)

```
SBFM <Wd>, <Wn>, #<immr>, #<imms>
```

64-bit (sf == 1 && N == 1)

```
SBFM <Xd>, <Xn>, #<immr>, #<imms>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

integer r;
integer s;
bits(datasize) wmask;
bits(datasize) tmask;

if sf == '1' && N != '1' then UNDEFINED;
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then UNDEFINED;

r = UInt(immr);
s = UInt(imms);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE, datasize);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<immr>	For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field. For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.
<imms>	For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field. For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

Alias Conditions

Alias	Of variant	Is preferred when
ASR (immediate)	32-bit	`imms == '011111'`
ASR (immediate)	64-bit	`imms == '111111'`

Alias	Of variant	Is preferred when
SBFIZ		`UInt(imms) < UInt(immr)`
SBFX		`BFXPreferred(sf, opc<1>, imms, immr)`
SXTB		`immr == '000000' && imms == '000111'`
SXTH		`immr == '000000' && imms == '001111'`
SXTW		`immr == '000000' && imms == '011111'`

Operation

```
bits(datasize) src = X[n, datasize];

// perform bitfield move on low bits
bits(datasize) bot = ROR(src, r) AND wmask;

// determine extension bits (sign, zero or dest register)
bits(datasize) top = Replicate(src<s>, datasize);

// combine extension bits and result bits
X[d, datasize] = (top AND NOT(tmask)) OR (bot AND tmask);
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

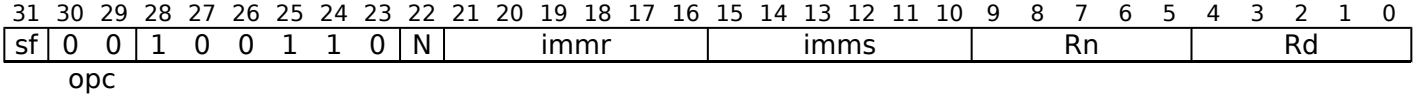
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SBFX

Signed Bitfield Extract copies a bitfield of <width> bits starting from bit position <lsb> in the source register to the least significant bits of the destination register, and sets destination bits above the bitfield to a copy of the most significant bit of the bitfield.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.



32-bit (sf == 0 && N == 0)

```
SBFX <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
SBFM <Wd>, <Wn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `BFXPreferred(sf, opc<1>, imms, immr)`.

64-bit (sf == 1 && N == 1)

```
SBFX <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
SBFM <Xd>, <Xn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `BFXPreferred(sf, opc<1>, imms, immr)`.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<lsb>	For the 32-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31. For the 64-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.
<width>	For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>. For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

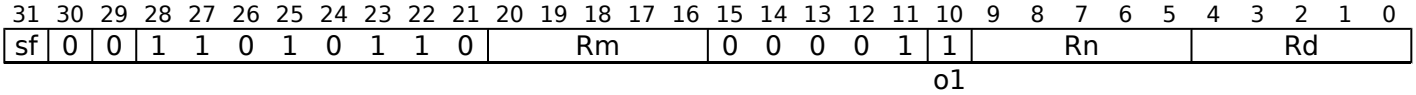
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

SDIV

Signed Divide divides a signed integer register value by another signed integer register value, and writes the result to the destination register. The condition flags are not affected.



32-bit (sf == 0)

```
SDIV <Wd>, <Wn>, <Wm>
```

64-bit (sf == 1)

```
SDIV <Xd>, <Xn>, <Xm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

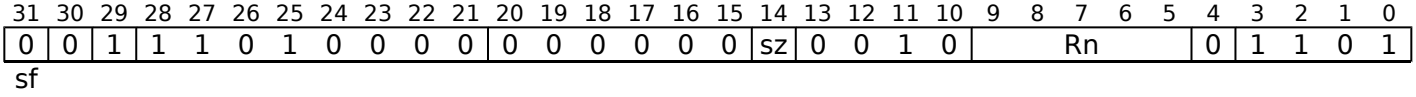
```
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = X[m, datasize];
integer result;

if IsZero(operand2) then
    result = 0;
else
    result = RoundTowardsZero(Real(Int(operand1, FALSE)) / Real(Int(operand2, FALSE)));

X[d, datasize] = result<datasize-1:0>;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SETF8, SETF16

Set the PSTATE.NZV flags based on the value in the specified general-purpose register. SETF8 treats the value as an 8 bit value, and SETF16 treats the value as an 16 bit value.
The PSTATE.C flag is not affected by these instructions.

Integer
(FEAT_FlagM)



SETF8 (sz == 0)

```
SETF8 <Wn>
```

SETF16 (sz == 1)

```
SETF16 <Wn>
```

```
if !HaveFlagManipulateExt() then UNDEFINED;
integer msb = if sz == '1' then 15 else 7;
integer n = UInt(Rn);
```

Assembler Symbols

<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
bits(32) tmpreg = X[n, 32];
PSTATE.N = tmpreg<msb>;
PSTATE.Z = if (tmpreg<msb:0> == Zeros(msb + 1)) then '1' else '0';
PSTATE.V = tmpreg<msb+1> EOR tmpreg<msb>;
//PSTATE.C unchanged;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SETGP, SETGM, SETGE

Memory Set with tag setting. These instructions perform a memory set using the value in the bottom byte of the source register and store an Allocation Tag to memory for each Tag Granule written. The Allocation Tag is calculated from the Logical Address Tag in the register which holds the first address that the set is made to. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETGP, then SETGM, and then SETGE.

SETGP performs some preconditioning of the arguments suitable for using the SETGM instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETGM performs an IMPLEMENTATION DEFINED amount of the memory set. SETGE performs the last part of the memory set.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETGP, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the set size is saturated to 0x7FFFFFFFFFFFFFF0.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETGP, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFF0.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETGM, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* number of bytes remaining to be set in the memory set in total.

For SETGM, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
    - the value of Xd is written back with the lowest address that has not been set.

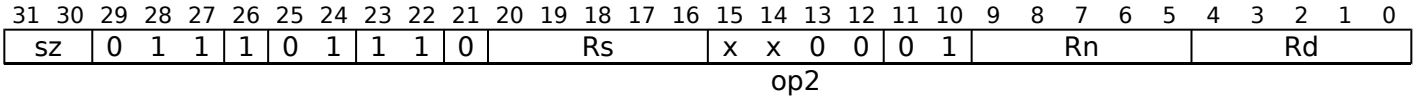
For SETGE, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1* the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For SETGE, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xd is written back with the lowest address that has not been set.

Integer
(FEAT_MOPS)



Epilogue (op2 == 1000)

```
SETGE [<Xd>]!, <Xn>!, <Xs>
```

Main (op2 == 0100)

```
SETGM [<Xd>]!, <Xn>!, <Xs>
```

Prologue (op2 == 0000)

```
SETGP [<Xd>]!, <Xn>!, <Xs>
```

```
if !HaveFeatMOPS() then UNDEFINED;
if !HaveMTEExt() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(2) options = op2<1:0>;

MOPSStage stage;
case op2<3:2> of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise UNDEFINED;

if s == n || s == d || n == d then UNDEFINED;
if d == 31 || n == 31 then UNDEFINED;
```

Assembler Symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and for option B is updated by the instruction, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and is updated by the instruction, encoded in the "Rd" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

<Xs> For the epilogue variant: is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

For the main and prologue variant: is the 64-bit name of the general-purpose register that holds the source data in bits<7:0>, encoded in the "Rs" field.

Operation

```
CheckMOPSEnabled();

bits(64) toaddress = X[d, 64];
bits(64) setsize = X[n, 64];
bits(8) data = X[s, 8];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagesetsize;
boolean is_setg = TRUE;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);

boolean supports_option_a = MemCpyOptionA();
acctype = MemSetAccessType(options);

if stage == MOPSStage_Prologue then
    if setsize<63> == '1' then setsize = 0x7FFFFFFFFFFFFFF0<63:0>;

    if setsize != Zeros(64) && toaddress != Align(toaddress, TAG_GRANULE) then
        boolean iswrite = TRUE;
        boolean secondstage = FALSE;
        AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

    if setsize != Align(setsize, TAG_GRANULE) then
        boolean iswrite = TRUE;
        boolean secondstage = FALSE;
        AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

    if supports_option_a then
        nzcv = '0000';
        toaddress = toaddress + setsize;
        setsize = Zeros(64) - setsize;
    else
        nzcv = '0010';

    stagesetsize = SETPreSizeChoice(toaddress, setsize, is_setg);
    assert stagesetsize<63> == setsize<63> || stagesetsize == Zeros(64);
    assert stagesetsize<3:0> == '0000';

    if SInt(setsize) > 0 then
        assert SInt(stagesetsize) <= SInt(setsize);
    else
        assert SInt(stagesetsize) >= SInt(setsize);
else
    bits(64) postsize = SETPostSizeChoice(toaddress, setsize, is_setg);
    assert postsize<63> == setsize<63> || postsize == Zeros(64);
    assert postsize<3:0> == '0000';

    boolean zero_size_exceptions = MemSetZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(setsize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, option
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, option

    if stage == MOPSStage_Main then
        stagesetsize = setsize - postsize;
        if MemSetParametersIllformedM(toaddress, setsize, is_setg) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, i
```

```
    else
        stagesetsize = postsize;
        if (setsize != postsize || MemSetParametersIllformedE(toaddress, setsize, is_setg)) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = TRUE;
            MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, i

    if setsize != Zeros(64) && toaddress != Align(toaddress, TAG_GRANULE) then
        boolean iswrite = TRUE;
        boolean secondstage = FALSE;
        AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

    if setsize != Align(setsize, TAG_GRANULE) then
        boolean iswrite = TRUE;
        boolean secondstage = FALSE;
        AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

integer tagstep;
bits(4) tag;
bits(64) tagaddr;

if supports_option_a then
    while SInt(stagesetsize) < 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 16);
        assert B <= -1 * SInt(stagesetsize);
        assert B<3:0> == '0000';

        Mem[toaddress+setsize, B, acctype] = Replicate(data, B);

        tagstep = B DIV 16;
        tag = AArch64.AllocationTagFromAddress(toaddress + setsize);
        while tagstep > 0 do
            tagaddr = toaddress + setsize + (tagstep - 1) * 16;
            AArch64.MemTag[tagaddr, acctype] = tag;
            tagstep = tagstep - 1;

        setsize = setsize + B;
        stagesetsize = stagesetsize + B;
        if stage != MOPSStage_Prologue then
            X[n, 64] = setsize;
else
    while UInt(stagesetsize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 16);
        assert B <= UInt(stagesetsize);
        assert B<3:0> == '0000';

        Mem[toaddress, B, acctype] = Replicate(data, B);

        tagstep = B DIV 16;
        tag = AArch64.AllocationTagFromAddress(toaddress);
        while tagstep > 0 do
            tagaddr = toaddress + (tagstep - 1) * 16;
            AArch64.MemTag[tagaddr, acctype] = tag;
            tagstep = tagstep - 1;

        toaddress = toaddress + B;
        setsize = setsize - B;
        stagesetsize = stagesetsize - B;
        if stage != MOPSStage_Prologue then
            X[n, 64] = setsize;
            X[d, 64] = toaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = setsize;
    X[d, 64] = toaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

SETGPN, SETGMN, SETGEN

Memory Set with tag setting, non-temporal. These instructions perform a memory set using the value in the bottom byte of the source register and store an Allocation Tag to memory for each Tag Granule written. The Allocation Tag is calculated from the Logical Address Tag in the register which holds the first address that the set is made to. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETGPN, then SETGMN, and then SETGEN.

SETGPN performs some preconditioning of the arguments suitable for using the SETGMN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETGMN performs an IMPLEMENTATION DEFINED amount of the memory set. SETGEN performs the last part of the memory set.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETGPN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the set size is saturated to 0x7FFFFFFFFFFFFFF0.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETGPN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFF0.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETGMN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* number of bytes remaining to be set in the memory set in total.

For SETGMN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
    - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
    - the value of Xd is written back with the lowest address that has not been set.

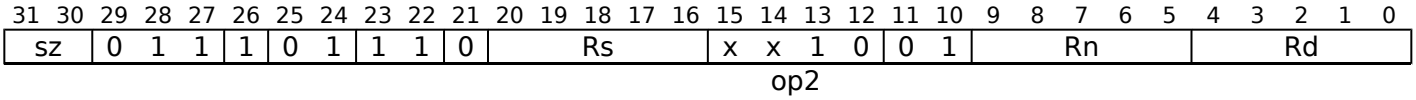
For SETGEN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1* the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For SETGEN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
    - the value of Xn is written back with 0.
    - the value of Xd is written back with the lowest address that has not been set.

Integer
(FEAT_MOPS)



Epilogue (op2 == 1010)

```
SETGEN [<Xd>]!, <Xn>!, <Xs>
```

Main (op2 == 0110)

```
SETGMN [<Xd>]!, <Xn>!, <Xs>
```

Prologue (op2 == 0010)

```
SETGPN [<Xd>]!, <Xn>!, <Xs>
```

```
if !HaveFeatMOPS() then UNDEFINED;
if !HaveMTEExt() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(2) options = op2<1:0>;

MOPSStage stage;
case op2<3:2> of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise UNDEFINED;

if s == n || s == d || n == d then UNDEFINED;
if d == 31 || n == 31 then UNDEFINED;
```

Assembler Symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and for option B is updated by the instruction, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and is updated by the instruction, encoded in the "Rd" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

<Xs> For the epilogue variant: is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

For the main and prologue variant: is the 64-bit name of the general-purpose register that holds the source data in bits<7:0>, encoded in the "Rs" field.

Operation

```
CheckMOPSEnabled();

bits(64) toaddress = X[d, 64];
bits(64) setsize = X[n, 64];
bits(8) data = X[s, 8];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagesetsize;
boolean is_setg = TRUE;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);

boolean supports_option_a = MemCpyOptionA();
acctype = MemSetAccessType(options);

if stage == MOPSStage_Prologue then
    if setsize<63> == '1' then setsize = 0x7FFFFFFFFFFFFFF0<63:0>;

    if setsize != Zeros(64) && toaddress != Align(toaddress, TAG_GRANULE) then
        boolean iswrite = TRUE;
        boolean secondstage = FALSE;
        AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

    if setsize != Align(setsize, TAG_GRANULE) then
        boolean iswrite = TRUE;
        boolean secondstage = FALSE;
        AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

    if supports_option_a then
        nzcv = '0000';
        toaddress = toaddress + setsize;
        setsize = Zeros(64) - setsize;
    else
        nzcv = '0010';

    stagesetsize = SETPreSizeChoice(toaddress, setsize, is_setg);
    assert stagesetsize<63> == setsize<63> || stagesetsize == Zeros(64);
    assert stagesetsize<3:0> == '0000';

    if SInt(setsize) > 0 then
        assert SInt(stagesetsize) <= SInt(setsize);
    else
        assert SInt(stagesetsize) >= SInt(setsize);
else
    bits(64) postsize = SETPostSizeChoice(toaddress, setsize, is_setg);
    assert postsize<63> == setsize<63> || postsize == Zeros(64);
    assert postsize<3:0> == '0000';

    boolean zero_size_exceptions = MemSetZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(setsize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, option
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, option

    if stage == MOPSStage_Main then
        stagesetsize = setsize - postsize;
        if MemSetParametersIllformedM(toaddress, setsize, is_setg) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, i
```

```
    else
        stagesetsize = postsize;
        if (setsize != postsize || MemSetParametersIllformedE(toaddress, setsize, is_setg)) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = TRUE;
            MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, i

    if setsize != Zeros(64) && toaddress != Align(toaddress, TAG_GRANULE) then
        boolean iswrite = TRUE;
        boolean secondstage = FALSE;
        AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

    if setsize != Align(setsize, TAG_GRANULE) then
        boolean iswrite = TRUE;
        boolean secondstage = FALSE;
        AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

integer tagstep;
bits(4) tag;
bits(64) tagaddr;

if supports_option_a then
    while SInt(stagesetsize) < 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 16);
        assert B <= -1 * SInt(stagesetsize);
        assert B<3:0> == '0000';

        Mem[toaddress+setsize, B, acctype] = Replicate(data, B);

        tagstep = B DIV 16;
        tag = AArch64.AllocationTagFromAddress(toaddress + setsize);
        while tagstep > 0 do
            tagaddr = toaddress + setsize + (tagstep - 1) * 16;
            AArch64.MemTag[tagaddr, acctype] = tag;
            tagstep = tagstep - 1;

        setsize = setsize + B;
        stagesetsize = stagesetsize + B;
        if stage != MOPSStage_Prologue then
            X[n, 64] = setsize;
else
    while UInt(stagesetsize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 16);
        assert B <= UInt(stagesetsize);
        assert B<3:0> == '0000';

        Mem[toaddress, B, acctype] = Replicate(data, B);

        tagstep = B DIV 16;
        tag = AArch64.AllocationTagFromAddress(toaddress);
        while tagstep > 0 do
            tagaddr = toaddress + (tagstep - 1) * 16;
            AArch64.MemTag[tagaddr, acctype] = tag;
            tagstep = tagstep - 1;

        toaddress = toaddress + B;
        setsize = setsize - B;
        stagesetsize = stagesetsize - B;
        if stage != MOPSStage_Prologue then
            X[n, 64] = setsize;
            X[d, 64] = toaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = setsize;
    X[d, 64] = toaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

SETGPT, SETGMT, SETGET

Memory Set with tag setting, unprivileged. These instructions perform a memory set using the value in the bottom byte of the source register and store an Allocation Tag to memory for each Tag Granule written. The Allocation Tag is calculated from the Logical Address Tag in the register which holds the first address that the set is made to. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETGPT, then SETGMT, and then SETGET.

SETGPT performs some preconditioning of the arguments suitable for using the SETGMT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETGMT performs an IMPLEMENTATION DEFINED amount of the memory set. SETGET performs the last part of the memory set.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETGPT, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the set size is saturated to 0x7FFFFFFFFFFFFFF0.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETGPT, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFF0.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETGMT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* number of bytes remaining to be set in the memory set in total.

For SETGMT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of Xd is written back with the lowest address that has not been set.

For SETGET, option A (encoded by PSTATE.C = 0), the format of the arguments is:

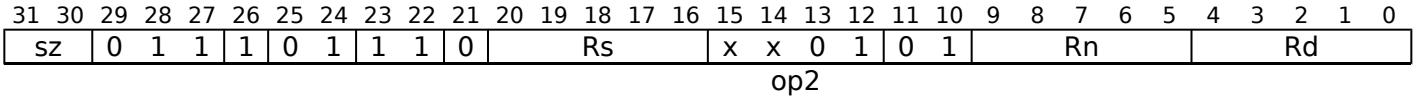
- Xn is treated as a signed 64-bit number.
- Xn holds -1* the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For SETGET, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xd is written back with the lowest address that has not been set.

Integer

(FEAT_MOPS)



Epilogue (op2 == 1001)

```
SETGET [<Xd>]!, <Xn>!, <Xs>
```

Main (op2 == 0101)

```
SETGMT [<Xd>]!, <Xn>!, <Xs>
```

Prologue (op2 == 0001)

```
SETGPT [<Xd>]!, <Xn>!, <Xs>
```

```
if !HaveFeatMOPS() then UNDEFINED;
if !HaveMTEExt() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(2) options = op2<1:0>;

MOPSStage stage;
case op2<3:2> of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise UNDEFINED;

if s == n || s == d || n == d then UNDEFINED;
if d == 31 || n == 31 then UNDEFINED;
```

Assembler Symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and for option B is updated by the instruction, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and is updated by the instruction, encoded in the "Rd" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

<Xs> For the epilogue variant: is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

For the main and prologue variant: is the 64-bit name of the general-purpose register that holds the source data in bits<7:0>, encoded in the "Rs" field.

Operation

```
CheckMOPSEnabled();

bits(64) toaddress = X[d, 64];
bits(64) setsize = X[n, 64];
bits(8) data = X[s, 8];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagesetsize;
boolean is_setg = TRUE;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);

boolean supports_option_a = MemCpyOptionA();
acctype = MemSetAccessType(options);

if stage == MOPSStage_Prologue then
    if setsize<63> == '1' then setsize = 0x7FFFFFFFFFFFFFF0<63:0>;

    if setsize != Zeros(64) && toaddress != Align(toaddress, TAG_GRANULE) then
        boolean iswrite = TRUE;
        boolean secondstage = FALSE;
        AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

    if setsize != Align(setsize, TAG_GRANULE) then
        boolean iswrite = TRUE;
        boolean secondstage = FALSE;
        AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

    if supports_option_a then
        nzcv = '0000';
        toaddress = toaddress + setsize;
        setsize = Zeros(64) - setsize;
    else
        nzcv = '0010';

    stagesetsize = SETPreSizeChoice(toaddress, setsize, is_setg);
    assert stagesetsize<63> == setsize<63> || stagesetsize == Zeros(64);
    assert stagesetsize<3:0> == '0000';

    if SInt(setsize) > 0 then
        assert SInt(stagesetsize) <= SInt(setsize);
    else
        assert SInt(stagesetsize) >= SInt(setsize);
else
    bits(64) postsize = SETPostSizeChoice(toaddress, setsize, is_setg);
    assert postsize<63> == setsize<63> || postsize == Zeros(64);
    assert postsize<3:0> == '0000';

    boolean zero_size_exceptions = MemSetZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(setsize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, option
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, option

    if stage == MOPSStage_Main then
        stagesetsize = setsize - postsize;
        if MemSetParametersIllformedM(toaddress, setsize, is_setg) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, i
```

```
    else
        stagesetsize = postsize;
        if (setsize != postsize || MemSetParametersIllformedE(toaddress, setsize, is_setg)) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = TRUE;
            MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, i

    if setsize != Zeros(64) && toaddress != Align(toaddress, TAG_GRANULE) then
        boolean iswrite = TRUE;
        boolean secondstage = FALSE;
        AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

    if setsize != Align(setsize, TAG_GRANULE) then
        boolean iswrite = TRUE;
        boolean secondstage = FALSE;
        AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

integer tagstep;
bits(4) tag;
bits(64) tagaddr;

if supports_option_a then
    while SInt(stagesetsize) < 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 16);
        assert B <= -1 * SInt(stagesetsize);
        assert B<3:0> == '0000';

        Mem[toaddress+setsize, B, acctype] = Replicate(data, B);

        tagstep = B DIV 16;
        tag = AArch64.AllocationTagFromAddress(toaddress + setsize);
        while tagstep > 0 do
            tagaddr = toaddress + setsize + (tagstep - 1) * 16;
            AArch64.MemTag[tagaddr, acctype] = tag;
            tagstep = tagstep - 1;

        setsize = setsize + B;
        stagesetsize = stagesetsize + B;
        if stage != MOPSStage_Prologue then
            X[n, 64] = setsize;
else
    while UInt(stagesetsize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 16);
        assert B <= UInt(stagesetsize);
        assert B<3:0> == '0000';

        Mem[toaddress, B, acctype] = Replicate(data, B);

        tagstep = B DIV 16;
        tag = AArch64.AllocationTagFromAddress(toaddress);
        while tagstep > 0 do
            tagaddr = toaddress + (tagstep - 1) * 16;
            AArch64.MemTag[tagaddr, acctype] = tag;
            tagstep = tagstep - 1;

        toaddress = toaddress + B;
        setsize = setsize - B;
        stagesetsize = stagesetsize - B;
        if stage != MOPSStage_Prologue then
            X[n, 64] = setsize;
            X[d, 64] = toaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = setsize;
    X[d, 64] = toaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

SETGPTN, SETGMTN, SETGETN

Memory Set with tag setting, unprivileged and non-temporal. These instructions perform a memory set using the value in the bottom byte of the source register and store an Allocation Tag to memory for each Tag Granule written. The Allocation Tag is calculated from the Logical Address Tag in the register which holds the first address that the set is made to. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETGPTN, then SETGMTN, and then SETGETN.

SETGPTN performs some preconditioning of the arguments suitable for using the SETGMTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETGMTN performs an IMPLEMENTATION DEFINED amount of the memory set. SETGETN performs the last part of the memory set.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETGPTN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the set size is saturated to 0x7FFFFFFFFFFFFFF0.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETGPTN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFF0.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETGMTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* number of bytes remaining to be set in the memory set in total.

For SETGMTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of Xd is written back with the lowest address that has not been set.

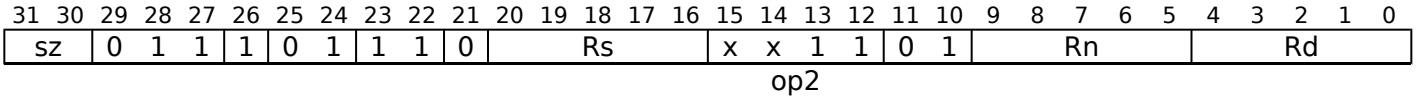
For SETGETN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1* the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For SETGETN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xd is written back with the lowest address that has not been set.

Integer
(FEAT_MOPS)



Epilogue (op2 == 1011)

```
SETGETN [<Xd>]!, <Xn>!, <Xs>
```

Main (op2 == 0111)

```
SETGMTN [<Xd>]!, <Xn>!, <Xs>
```

Prologue (op2 == 0011)

```
SETGPTN [<Xd>]!, <Xn>!, <Xs>
```

```
if !HaveFeatMOPS() then UNDEFINED;
if !HaveMTEExt() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(2) options = op2<1:0>;

MOPSStage stage;
case op2<3:2> of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise UNDEFINED;

if s == n || s == d || n == d then UNDEFINED;
if d == 31 || n == 31 then UNDEFINED;
```

Assembler Symbols

<Xd> For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and for option B is updated by the instruction, encoded in the "Rd" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address (an integer multiple of 16) and is updated by the instruction, encoded in the "Rd" field.

<Xn> For the epilogue variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is set to zero at the end of the instruction, encoded in the "Rn" field.

For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set (an integer multiple of 16) and is updated by the instruction, encoded in the "Rn" field.

<Xs> For the epilogue variant: is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

For the main and prologue variant: is the 64-bit name of the general-purpose register that holds the source data in bits<7:0>, encoded in the "Rs" field.

Operation

```
CheckMOPSEnabled();

bits(64) toaddress = X[d, 64];
bits(64) setsize = X[n, 64];
bits(8) data = X[s, 8];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagesetsize;
boolean is_setg = TRUE;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);

boolean supports_option_a = MemCpyOptionA();
acctype = MemSetAccessType(options);

if stage == MOPSStage_Prologue then
    if setsize<63> == '1' then setsize = 0x7FFFFFFFFFFFFFF0<63:0>;

    if setsize != Zeros(64) && toaddress != Align(toaddress, TAG_GRANULE) then
        boolean iswrite = TRUE;
        boolean secondstage = FALSE;
        AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

    if setsize != Align(setsize, TAG_GRANULE) then
        boolean iswrite = TRUE;
        boolean secondstage = FALSE;
        AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

    if supports_option_a then
        nzcv = '0000';
        toaddress = toaddress + setsize;
        setsize = Zeros(64) - setsize;
    else
        nzcv = '0010';

    stagesetsize = SETPreSizeChoice(toaddress, setsize, is_setg);
    assert stagesetsize<63> == setsize<63> || stagesetsize == Zeros(64);
    assert stagesetsize<3:0> == '0000';

    if SInt(setsize) > 0 then
        assert SInt(stagesetsize) <= SInt(setsize);
    else
        assert SInt(stagesetsize) >= SInt(setsize);
else
    bits(64) postsize = SETPostSizeChoice(toaddress, setsize, is_setg);
    assert postsize<63> == setsize<63> || postsize == Zeros(64);
    assert postsize<3:0> == '0000';

    boolean zero_size_exceptions = MemSetZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(setsize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, option
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, option

    if stage == MOPSStage_Main then
        stagesetsize = setsize - postsize;
        if MemSetParametersIllformedM(toaddress, setsize, is_setg) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, i
```

```
    else
        stagesetsize = postsize;
        if (setsize != postsize || MemSetParametersIllformedE(toaddress, setsize, is_setg)) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = TRUE;
            MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, i

    if setsize != Zeros(64) && toaddress != Align(toaddress, TAG_GRANULE) then
        boolean iswrite = TRUE;
        boolean secondstage = FALSE;
        AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

    if setsize != Align(setsize, TAG_GRANULE) then
        boolean iswrite = TRUE;
        boolean secondstage = FALSE;
        AArch64.Abort(toaddress, AlignmentFault(acctype, iswrite, secondstage));

integer tagstep;
bits(4) tag;
bits(64) tagaddr;

if supports_option_a then
    while SInt(stagesetsize) < 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 16);
        assert B <= -1 * SInt(stagesetsize);
        assert B<3:0> == '0000';

        Mem[toaddress+setsize, B, acctype] = Replicate(data, B);

        tagstep = B DIV 16;
        tag = AArch64.AllocationTagFromAddress(toaddress + setsize);
        while tagstep > 0 do
            tagaddr = toaddress + setsize + (tagstep - 1) * 16;
            AArch64.MemTag[tagaddr, acctype] = tag;
            tagstep = tagstep - 1;

        setsize = setsize + B;
        stagesetsize = stagesetsize + B;
        if stage != MOPSStage_Prologue then
            X[n, 64] = setsize;
else
    while UInt(stagesetsize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 16);
        assert B <= UInt(stagesetsize);
        assert B<3:0> == '0000';

        Mem[toaddress, B, acctype] = Replicate(data, B);

        tagstep = B DIV 16;
        tag = AArch64.AllocationTagFromAddress(toaddress);
        while tagstep > 0 do
            tagaddr = toaddress + (tagstep - 1) * 16;
            AArch64.MemTag[tagaddr, acctype] = tag;
            tagstep = tagstep - 1;

        toaddress = toaddress + B;
        setsize = setsize - B;
        stagesetsize = stagesetsize - B;
        if stage != MOPSStage_Prologue then
            X[n, 64] = setsize;
            X[d, 64] = toaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = setsize;
    X[d, 64] = toaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

SETP, SETM, SETE

Memory Set. These instructions perform a memory set using the value in the bottom byte of the source register. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETP, then SETM, and then SETE.

SETP performs some preconditioning of the arguments suitable for using the SETM instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETM performs an IMPLEMENTATION DEFINED amount of the memory set. SETE performs the last part of the memory set.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETP, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the set size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETP, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETM, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be set in the memory set in total.

For SETM, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of Xd is written back with the lowest address that has not been set.

For SETE, option A (encoded by PSTATE.C = 0), the format of the arguments is:

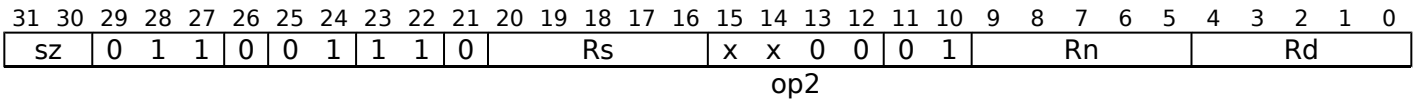
- Xn is treated as a signed 64-bit number.
- Xn holds -1* the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For SETE, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xd is written back with the lowest address that has not been set.

Integer

(FEAT_MOPS)



Epilogue (op2 == 1000)

```
SETE [<Xd>]!, <Xn>!, <Xs>
```

Main (op2 == 0100)

```
SETM [<Xd>]!, <Xn>!, <Xs>
```

Prologue (op2 == 0000)

```
SETP [<Xd>]!, <Xn>!, <Xs>
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(2) options = op2<1:0>;

MOPSStage stage;
case op2<3:2> of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise UNDEFINED;

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || n == 31 then
    c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address and for option B is updated by the instruction, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.
<Xs>	Is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

Operation

```
bits(64) toaddress = X[d, 64];
bits(64) setsize = X[n, 64];
bits(8) data = X[s, 8];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagesetsize;
boolean is_setg = FALSE;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
acctype = MemSetAccessType(options);

if stage == MOPSStage_Prologue then
    if setsize<63> == '1' then setsize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if supports_option_a then
        nzcv = '0000';
        toaddress = toaddress + setsize;
        setsize = Zeros(64) - setsize;
    else
        nzcv = '0010';

    stagesetsize = SETPreSizeChoice(toaddress, setsize, is_setg);
    assert stagesetsize<63> == setsize<63> || stagesetsize == Zeros(64);

    if SInt(setsize) > 0 then
        assert SInt(stagesetsize) <= SInt(setsize);
    else
        assert SInt(stagesetsize) >= SInt(setsize);
else
    bits(64) postsize = SETPostSizeChoice(toaddress, setsize, is_setg);
    assert postsize<63> == setsize<63> || postsize == Zeros(64);

    boolean zero_size_exceptions = MemSetZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(setsize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, option
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, option

    if stage == MOPSStage_Main then
        stagesetsize = setsize - postsize;
        if MemSetParametersIllformedM(toaddress, setsize, is_setg) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, i
    else
        stagesetsize = postsize;
        if (setsize != postsize || MemSetParametersIllformedE(toaddress, setsize, is_setg)) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = TRUE;
            MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, i

if supports_option_a then
    while SInt(stagesetsize) < 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 1);
        assert B <= -1 * SInt(stagesetsize);
```

```
        Mem[toaddress+setsize, B, acctype] = Replicate(data, B);
        setsize = setsize + B;
        stagesetsize = stagesetsize + B;
        if stage != MOPSStage_Prologue then
            X[n, 64] = setsize;
else
    while UInt(stagesetsize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 1);
        assert B <= UInt(stagesetsize);

        Mem[toaddress, B, acctype] = Replicate(data, B);
        toaddress = toaddress + B;
        setsize = setsize - B;
        stagesetsize = stagesetsize - B;
        if stage != MOPSStage_Prologue then
            X[n, 64] = setsize;
            X[d, 64] = toaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = setsize;
    X[d, 64] = toaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SETPN, SETMN, SETEN

Memory Set, non-temporal. These instructions perform a memory set using the value in the bottom byte of the source register. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETPN, then SETMN, and then SETEN.

SETPN performs some preconditioning of the arguments suitable for using the SETMN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETMN performs an IMPLEMENTATION DEFINED amount of the memory set. SETEN performs the last part of the memory set.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETPN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the set size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETPN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETMN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be set in the memory set in total.

For SETMN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of Xd is written back with the lowest address that has not been set.

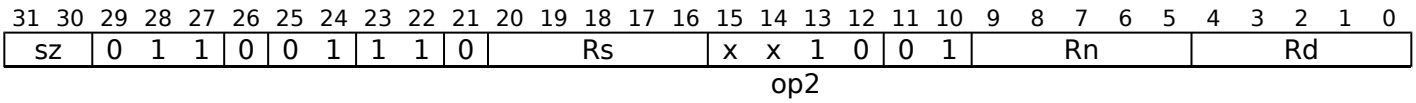
For SETEN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1* the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For SETEN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xd is written back with the lowest address that has not been set.

Integer
(FEAT_MOPS)



Epilogue (op2 == 1010)

```
SETEN [<Xd>]!, <Xn>!, <Xs>
```

Main (op2 == 0110)

```
SETMN [<Xd>]!, <Xn>!, <Xs>
```

Prologue (op2 == 0010)

```
SETPN [<Xd>]!, <Xn>!, <Xs>
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(2) options = op2<1:0>;

MOPSStage stage;
case op2<3:2> of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise UNDEFINED;

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || n == 31 then
    c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address and for option B is updated by the instruction, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.
<Xs>	Is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

Operation

```
bits(64) toaddress = X[d, 64];
bits(64) setsize = X[n, 64];
bits(8) data = X[s, 8];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagesetsize;
boolean is_setg = FALSE;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
acctype = MemSetAccessType(options);

if stage == MOPSStage_Prologue then
    if setsize<63> == '1' then setsize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if supports_option_a then
        nzcv = '0000';
        toaddress = toaddress + setsize;
        setsize = Zeros(64) - setsize;
    else
        nzcv = '0010';

    stagesetsize = SETPreSizeChoice(toaddress, setsize, is_setg);
    assert stagesetsize<63> == setsize<63> || stagesetsize == Zeros(64);

    if SInt(setsize) > 0 then
        assert SInt(stagesetsize) <= SInt(setsize);
    else
        assert SInt(stagesetsize) >= SInt(setsize);
else
    bits(64) postsize = SETPostSizeChoice(toaddress, setsize, is_setg);
    assert postsize<63> == setsize<63> || postsize == Zeros(64);

    boolean zero_size_exceptions = MemSetZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(setsize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, option
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, option

    if stage == MOPSStage_Main then
        stagesetsize = setsize - postsize;
        if MemSetParametersIllformedM(toaddress, setsize, is_setg) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, i
    else
        stagesetsize = postsize;
        if (setsize != postsize || MemSetParametersIllformedE(toaddress, setsize, is_setg)) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = TRUE;
            MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, i

if supports_option_a then
    while SInt(stagesetsize) < 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 1);
        assert B <= -1 * SInt(stagesetsize);
```

```
        Mem[toaddress+setsize, B, acctype] = Replicate(data, B);
        setsize = setsize + B;
        stagesetsize = stagesetsize + B;
        if stage != MOPSStage_Prologue then
            X[n, 64] = setsize;
else
    while UInt(stagesetsize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 1);
        assert B <= UInt(stagesetsize);

        Mem[toaddress, B, acctype] = Replicate(data, B);
        toaddress = toaddress + B;
        setsize = setsize - B;
        stagesetsize = stagesetsize - B;
        if stage != MOPSStage_Prologue then
            X[n, 64] = setsize;
            X[d, 64] = toaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = setsize;
    X[d, 64] = toaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SETPT, SETMT, SETET

Memory Set, unprivileged. These instructions perform a memory set using the value in the bottom byte of the source register. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETPT, then SETMT, and then SETET.

SETPT performs some preconditioning of the arguments suitable for using the SETMT instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETMT performs an IMPLEMENTATION DEFINED amount of the memory set. SETET performs the last part of the memory set.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETPT, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the set size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETPT, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETMT, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be set in the memory set in total.

For SETMT, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of Xd is written back with the lowest address that has not been set.

For SETET, option A (encoded by PSTATE.C = 0), the format of the arguments is:

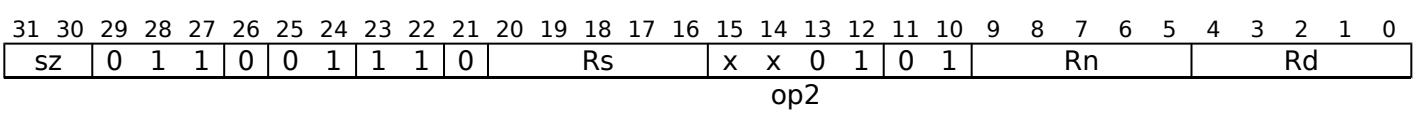
- Xn is treated as a signed 64-bit number.
- Xn holds -1* the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For SETET, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xd is written back with the lowest address that has not been set.

Integer

(FEAT_MOPS)



Epilogue (op2 == 1001)

```
SETET [<Xd>]!, <Xn>!, <Xs>
```

Main (op2 == 0101)

```
SETMT [<Xd>]!, <Xn>!, <Xs>
```

Prologue (op2 == 0001)

```
SETPT [<Xd>]!, <Xn>!, <Xs>
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(2) options = op2<1:0>;

MOPSStage stage;
case op2<3:2> of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise UNDEFINED;

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || n == 31 then
    c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address and for option B is updated by the instruction, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.
<Xs>	Is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

Operation

```
bits(64) toaddress = X[d, 64];
bits(64) setsize = X[n, 64];
bits(8) data = X[s, 8];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagesetsize;
boolean is_setg = FALSE;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
acctype = MemSetAccessType(options);

if stage == MOPSStage_Prologue then
    if setsize<63> == '1' then setsize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if supports_option_a then
        nzcv = '0000';
        toaddress = toaddress + setsize;
        setsize = Zeros(64) - setsize;
    else
        nzcv = '0010';

    stagesetsize = SETPreSizeChoice(toaddress, setsize, is_setg);
    assert stagesetsize<63> == setsize<63> || stagesetsize == Zeros(64);

    if SInt(setsize) > 0 then
        assert SInt(stagesetsize) <= SInt(setsize);
    else
        assert SInt(stagesetsize) >= SInt(setsize);
else
    bits(64) postsize = SETPostSizeChoice(toaddress, setsize, is_setg);
    assert postsize<63> == setsize<63> || postsize == Zeros(64);

    boolean zero_size_exceptions = MemSetZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(setsize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, option
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, option

    if stage == MOPSStage_Main then
        stagesetsize = setsize - postsize;
        if MemSetParametersIllformedM(toaddress, setsize, is_setg) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, i
    else
        stagesetsize = postsize;
        if (setsize != postsize || MemSetParametersIllformedE(toaddress, setsize, is_setg)) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = TRUE;
            MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, i

if supports_option_a then
    while SInt(stagesetsize) < 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 1);
        assert B <= -1 * SInt(stagesetsize);
```

```
        Mem[toaddress+setsize, B, acctype] = Replicate(data, B);
        setsize = setsize + B;
        stagesetsize = stagesetsize + B;
        if stage != MOPSStage_Prologue then
            X[n, 64] = setsize;
else
    while UInt(stagesetsize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 1);
        assert B <= UInt(stagesetsize);

        Mem[toaddress, B, acctype] = Replicate(data, B);
        toaddress = toaddress + B;
        setsize = setsize - B;
        stagesetsize = stagesetsize - B;
        if stage != MOPSStage_Prologue then
            X[n, 64] = setsize;
            X[d, 64] = toaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = setsize;
    X[d, 64] = toaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SETPTN, SETMTN, SETETN

Memory Set, unprivileged and non-temporal. These instructions perform a memory set using the value in the bottom byte of the source register. The prologue, main, and epilogue instructions are expected to be run in succession and to appear consecutively in memory: SETPTN, then SETMTN, and then SETETN.

SETPTN performs some preconditioning of the arguments suitable for using the SETMTN instruction, and performs an IMPLEMENTATION DEFINED amount of the memory set. SETMTN performs an IMPLEMENTATION DEFINED amount of the memory set. SETETN performs the last part of the memory set.

Note

The inclusion of IMPLEMENTATION DEFINED amounts of memory set allows some optimization of the size that can be performed.

The architecture supports two algorithms for the memory set: option A and option B. Which algorithm is used is IMPLEMENTATION DEFINED.

Note

Portable software should not assume that the choice of algorithm is constant.

After execution of SETPTN, option A (which results in encoding PSTATE.C = 0):

- If Xn<63> == 1, the set size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xd holds the original Xd + saturated Xn.
- Xn holds -1* saturated Xn + an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

After execution of SETPTN, option B (which results in encoding PSTATE.C = 1):

- If Xn<63> == 1, the copy size is saturated to 0x7FFFFFFFFFFFFFFF.
- Xd holds the original Xd + an IMPLEMENTATION DEFINED number of bytes set.
- Xn holds the saturated Xn - an IMPLEMENTATION DEFINED number of bytes set.
- PSTATE.{N,Z,V} are set to {0,0,0}.

For SETMTN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1* number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with -1* the number of bytes remaining to be set in the memory set in total.

For SETMTN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with the number of bytes remaining to be set in the memory set in total.
  - the value of Xd is written back with the lowest address that has not been set.

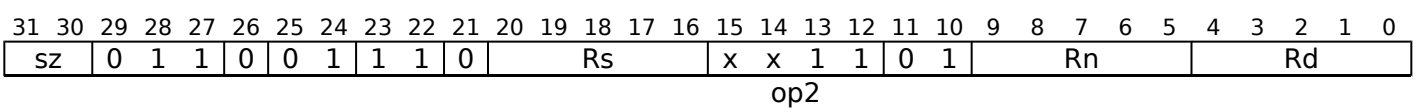
For SETETN, option A (encoded by PSTATE.C = 0), the format of the arguments is:

- Xn is treated as a signed 64-bit number.
- Xn holds -1* the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to -Xn.
- At the end of the instruction, the value of Xn is written back with 0.

For SETETN, option B (encoded by PSTATE.C = 1), the format of the arguments is:

- Xn holds the number of bytes remaining to be set in the memory set in total.
- Xd holds the lowest address that the set is made to.
- At the end of the instruction:
  - the value of Xn is written back with 0.
  - the value of Xd is written back with the lowest address that has not been set.

Integer
(FEAT_MOPS)



Epilogue (op2 == 1011)

```
SETETN [<Xd>]!, <Xn>!, <Xs>
```

Main (op2 == 0111)

```
SETMTN [<Xd>]!, <Xn>!, <Xs>
```

Prologue (op2 == 0011)

```
SETPTN [<Xd>]!, <Xn>!, <Xs>
```

```
if !HaveFeatMOPS() then UNDEFINED;
if sz != '00' then UNDEFINED;

integer d = UInt(Rd);
integer s = UInt(Rs);
integer n = UInt(Rn);
bits(2) options = op2<1:0>;

MOPSStage stage;
case op2<3:2> of
    when '00' stage = MOPSStage_Prologue;
    when '01' stage = MOPSStage_Main;
    when '10' stage = MOPSStage_Epilogue;
    otherwise UNDEFINED;

CheckMOPSEnabled();

if s == n || s == d || n == d || d == 31 || n == 31 then
    c = ConstrainUnpredictable(Unpredictable_MOPSOVERLAP31);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF UNDEFINED;
        when Constraint_NOP   EndOfInstruction();
```

Assembler Symbols

<Xd>	For the epilogue and main variant: is the 64-bit name of the general-purpose register that holds an encoding of the destination address and for option B is updated by the instruction, encoded in the "Rd" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the destination address and is updated by the instruction, encoded in the "Rd" field.
<Xn>	For the epilogue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is set to zero at the end of the instruction, encoded in the "Rn" field. For the main variant: is the 64-bit name of the general-purpose register that holds an encoding of the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field. For the prologue variant: is the 64-bit name of the general-purpose register that holds the number of bytes to be set and is updated by the instruction, encoded in the "Rn" field.
<Xs>	Is the 64-bit name of the general-purpose register that holds the source data, encoded in the "Rs" field.

Operation

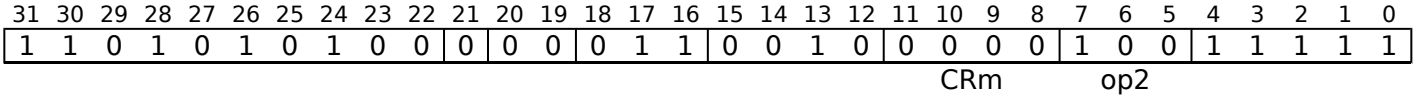
```
bits(64) toaddress = X[d, 64];
bits(64) setsize = X[n, 64];
bits(8) data = X[s, 8];
bits(4) nzcv = PSTATE.<N,Z,C,V>;
bits(64) stagesetsize;
boolean is_setg = FALSE;
integer B;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

boolean supports_option_a = MemCpyOptionA();
acctype = MemSetAccessType(options);

if stage == MOPSStage_Prologue then
    if setsize<63> == '1' then setsize = 0x7FFFFFFFFFFFFFFF<63:0>;

    if supports_option_a then
        nzcv = '0000';
        toaddress = toaddress + setsize;
        setsize = Zeros(64) - setsize;
    else
        nzcv = '0010';

    stagesetsize = SETPreSizeChoice(toaddress, setsize, is_setg);
    assert stagesetsize<63> == setsize<63> || stagesetsize == Zeros(64);

    if SInt(setsize) > 0 then
        assert SInt(stagesetsize) <= SInt(setsize);
    else
        assert SInt(stagesetsize) >= SInt(setsize);
else
    bits(64) postsize = SETPostSizeChoice(toaddress, setsize, is_setg);
    assert postsize<63> == setsize<63> || postsize == Zeros(64);

    boolean zero_size_exceptions = MemSetZeroSizeCheck();

    // Check if this version is consistent with the state of the call.
    if zero_size_exceptions || SInt(setsize) != 0 then
        if supports_option_a then
            if nzcv<1> == '1' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, option
        else
            if nzcv<1> == '0' then // PSTATE.C
                boolean wrong_option = TRUE;
                boolean from_epilogue = stage == MOPSStage_Epilogue;
                MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, option

    if stage == MOPSStage_Main then
        stagesetsize = setsize - postsize;
        if MemSetParametersIllformedM(toaddress, setsize, is_setg) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = FALSE;
            MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, i
    else
        stagesetsize = postsize;
        if (setsize != postsize || MemSetParametersIllformedE(toaddress, setsize, is_setg)) then
            boolean wrong_option = FALSE;
            boolean from_epilogue = TRUE;
            MismatchedMemSetException(supports_option_a, d, s, n, wrong_option, from_epilogue, options, i

if supports_option_a then
    while SInt(stagesetsize) < 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 1);
        assert B <= -1 * SInt(stagesetsize);
```

```
        Mem[toaddress+setsize, B, acctype] = Replicate(data, B);
        setsize = setsize + B;
        stagesetsize = stagesetsize + B;
        if stage != MOPSStage_Prologue then
            X[n, 64] = setsize;
else
    while UInt(stagesetsize) > 0 do
        // IMP DEF selection of the block size that is worked on. While many
        // implementations might make this constant, that is not assumed.
        B = SETSizeChoice(toaddress, setsize, 1);
        assert B <= UInt(stagesetsize);

        Mem[toaddress, B, acctype] = Replicate(data, B);
        toaddress = toaddress + B;
        setsize = setsize - B;
        stagesetsize = stagesetsize - B;
        if stage != MOPSStage_Prologue then
            X[n, 64] = setsize;
            X[d, 64] = toaddress;

if stage == MOPSStage_Prologue then
    X[n, 64] = setsize;
    X[d, 64] = toaddress;
    PSTATE.<N,Z,C,V> = nzcv;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SEV

Send Event is a hint instruction. It causes an event to be signaled to all PEs in the multiprocessor system. For more information, see *Wait for Event mechanism and Send event*.



SEV

```
// Empty.
```

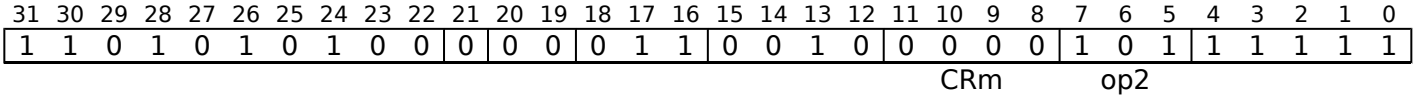
Operation

```
SendEvent();
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SEVL

Send Event Local is a hint instruction that causes an event to be signaled locally without requiring the event to be signaled to other PEs in the multiprocessor system. It can prime a wait-loop which starts with a `WFE` instruction.



SEVL

```
// Empty.
```

Operation

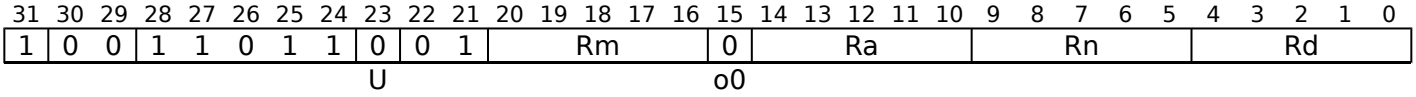
```
SendEventLocal();
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMADDL

Signed Multiply-Add Long multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias SMULL.



SMADDL <Xd>, <Wn>, <Wm>, <Xa>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
```

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Xa>	Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

Alias Conditions

Alias	Is preferred when
SMULL	`Ra == '11111'`

Operation

```
bits(32) operand1 = X[n, 32];
bits(32) operand2 = X[m, 32];
bits(64) operand3 = X[a, 64];

integer result;

result = Int(operand3, FALSE) + (Int(operand1, FALSE) * Int(operand2, FALSE));

X[d, 64] = result<63:0>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMC

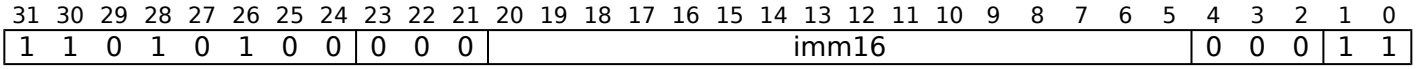
Secure Monitor Call causes an exception to EL3.

SMC is available only for software executing at EL1 or higher. It is UNDEFINED in EL0.

If the values of *HCR_EL2*.TSC and *SCR_EL3*.SMD are both 0, execution of an SMC instruction at EL1 or higher generates a Secure Monitor Call exception, recording it in *ESR_ELx*, using the EC value 0x17, that is taken to EL3.

If the value of *HCR_EL2*.TSC is 1 and EL2 is enabled in the current Security state, execution of an SMC instruction at EL1 generates an exception that is taken to EL2, regardless of the value of *SCR_EL3*.SMD.

If the value of *HCR_EL2*.TSC is 0 and the value of *SCR_EL3*.SMD is 1, the SMC instruction is UNDEFINED.



SMC #<imm>

```
// Empty.
```

Assembler Symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation

```
AArch64.CheckForSMCUndefOrTrap(imm16);
AArch64.CallSecureMonitor(imm16);
```

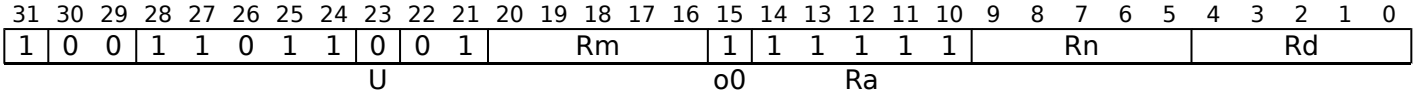
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMNEGL

Signed Multiply-Negate Long multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

This is an alias of SMSUBL. This means:

- The encodings in this description are named to match the encodings of SMSUBL.
- The description of SMSUBL gives the operational pseudocode for this instruction.



SMNEGL <Xd>, <Wn>, <Wm>

is equivalent to

SMSUBL <Xd>, <Wn>, <Wm>, XZR

and is always the preferred disassembly.

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of SMSUBL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

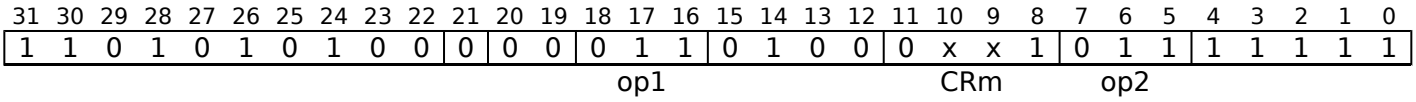
SMSTART

Enables access to Streaming SVE mode and SME architectural state.
SMSTART enters Streaming SVE mode, and enables the SME ZA storage.
SMSTART SM enters Streaming SVE mode, but does not enable the SME ZA storage.
SMSTART ZA enables the SME ZA storage, but does not cause an entry to Streaming SVE mode.

This is an alias of MSR (immediate). This means:

- The encodings in this description are named to match the encodings of MSR (immediate).
- The description of MSR (immediate) gives the operational pseudocode for this instruction.

System
(FEAT_SME)



SMSTART {<option>}

is equivalent to

MSR <pstatefield>, #1

and is always the preferred disassembly.

Assembler Symbols

<option> Is an optional mode, encoded in “CRm<2:1>”:

CRm<2:1>	<option>
00	RESERVED
01	SM
10	ZA
11	[no specifier]

<pstatefield> Is a PSTATE field name. For the MSR instruction, this is encoded in “op1:op2:CRm”:

op1	op2	CRm	<pstatefield>	Architectural Feature
000	00x	xxxx	SEE_PSTATE	-
000	010	xxxx	SEE_PSTATE	-
000	011	xxxx	UAO	FEAT_UAO
000	100	xxxx	PAN	FEAT_PAN
000	101	xxxx	SPSel	-
000	11x	xxxx	RESERVED	-
001	000	000x	ALLINT	FEAT_NMI
001	000	001x	RESERVED	-
001	000	01xx	RESERVED	-
001	000	1xxx	RESERVED	-
001	001	xxxx	RESERVED	-
001	01x	xxxx	RESERVED	-
001	1xx	xxxx	RESERVED	-
010	xxx	xxxx	RESERVED	-
011	000	xxxx	RESERVED	-
011	001	xxxx	SSBS	FEAT_SSBS
011	010	xxxx	DIT	FEAT_DIT
011	011	000x	RESERVED	-
011	011	001x	SVCRSM	FEAT_SME
011	011	010x	SVCRZA	FEAT_SME
011	011	011x	SVCRSMZA	FEAT_SME
011	011	1xxx	RESERVED	-
011	100	xxxx	TCO	FEAT_MTE
011	101	xxxx	RESERVED	-
011	110	xxxx	DAIFSet	-
011	111	xxxx	DAIFClr	-
1xx	xxx	xxxx	RESERVED	-

For the SMSTART and SMSTOP aliases, this is encoded in "CRm<2:1>", where 0b01 specifies SVCRSM, 0b10 specifies SVCRZA, and 0b11 specifies SVCRSMZA.

Operation

The description of MSR (immediate) gives the operational pseudocode for this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

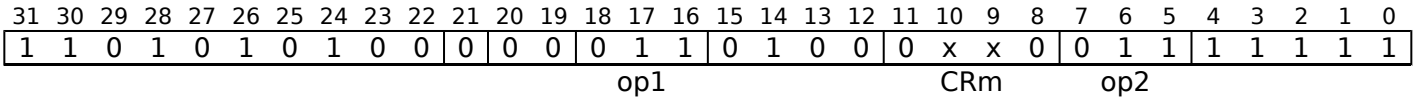
SMSTOP

Disables access to Streaming SVE mode and SME architectural state.
SMSTOP exits Streaming SVE mode, and disables the SME ZA storage.
SMSTOP SM exits Streaming SVE mode, but does not disable the SME ZA storage.
SMSTOP ZA disables the SME ZA storage, but does not cause an exit from Streaming SVE mode.

This is an alias of MSR (immediate). This means:

- The encodings in this description are named to match the encodings of MSR (immediate).
- The description of MSR (immediate) gives the operational pseudocode for this instruction.

System
(FEAT_SME)



SMSTOP {<option>}

is equivalent to

MSR <pstatefield>, #0

and is always the preferred disassembly.

Assembler Symbols

<option> Is an optional mode, encoded in “CRm<2:1>”:

CRm<2:1>	<option>
00	RESERVED
01	SM
10	ZA
11	[no specifier]

<pstatefield> Is a PSTATE field name. For the MSR instruction, this is encoded in “op1:op2:CRm”:

op1	op2	CRm	<pstatefield>	Architectural Feature
000	00x	xxxx	SEE_PSTATE	-
000	010	xxxx	SEE_PSTATE	-
000	011	xxxx	UAO	FEAT_UAO
000	100	xxxx	PAN	FEAT_PAN
000	101	xxxx	SPSel	-
000	11x	xxxx	RESERVED	-
001	000	000x	ALLINT	FEAT_NMI
001	000	001x	RESERVED	-
001	000	01xx	RESERVED	-
001	000	1xxx	RESERVED	-
001	001	xxxx	RESERVED	-
001	01x	xxxx	RESERVED	-
001	1xx	xxxx	RESERVED	-
010	xxx	xxxx	RESERVED	-
011	000	xxxx	RESERVED	-
011	001	xxxx	SSBS	FEAT_SSBS
011	010	xxxx	DIT	FEAT_DIT
011	011	000x	RESERVED	-
011	011	001x	SVCRSM	FEAT_SME
011	011	010x	SVCRZA	FEAT_SME
011	011	011x	SVCRSMZA	FEAT_SME
011	011	1xxx	RESERVED	-
011	100	xxxx	TCO	FEAT_MTE
011	101	xxxx	RESERVED	-
011	110	xxxx	DAIFSet	-
011	111	xxxx	DAIFClr	-
1xx	xxx	xxxx	RESERVED	-

For the SMSTART and SMSTOP aliases, this is encoded in "CRm<2:1>", where 0b01 specifies SVCRSM, 0b10 specifies SVCRZA, and 0b11 specifies SVCRSMZA.

Operation

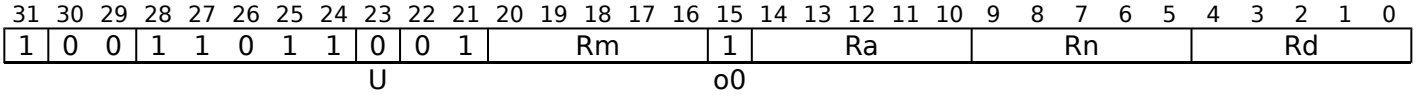
The description of MSR (immediate) gives the operational pseudocode for this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMSUBL

Signed Multiply-Subtract Long multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias SMNEGL.



SMSUBL <Xd>, <Wn>, <Wm>, <Xa>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
```

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Xa>	Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

Alias Conditions

Alias	Is preferred when
SMNEGL	`Ra == '11111'`

Operation

```
bits(32) operand1 = X[n, 32];
bits(32) operand2 = X[m, 32];
bits(64) operand3 = X[a, 64];

integer result;

result = Int(operand3, FALSE) - (Int(operand1, FALSE) * Int(operand2, FALSE));
X[d, 64] = result<63:0>;
```

Operational information

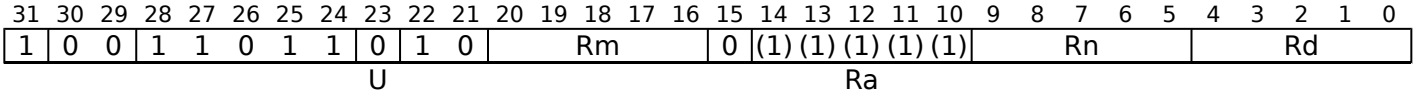
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMULH

Signed Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.



SMULH <Xd>, <Xn>, <Xm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

```
bits(64) operand1 = X[n, 64];
bits(64) operand2 = X[m, 64];

integer result;

result = Int(operand1, FALSE) * Int(operand2, FALSE);

X[d, 64] = result<127:64>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

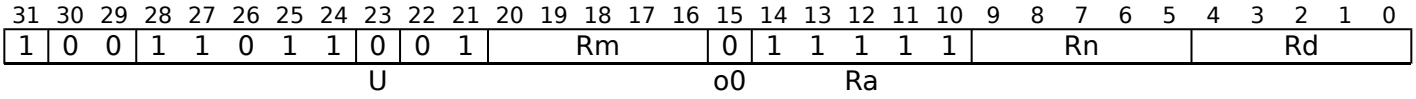
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMULL

Signed Multiply Long multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

This is an alias of SMADDL. This means:

- The encodings in this description are named to match the encodings of SMADDL.
- The description of SMADDL gives the operational pseudocode for this instruction.



SMULL <Xd>, <Wn>, <Wm>

is equivalent to

SMADDL <Xd>, <Wn>, <Wm>, XZR

and is always the preferred disassembly.

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of SMADDL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SSBB

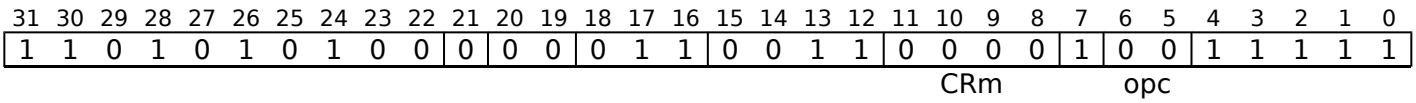
Speculative Store Bypass Barrier is a memory barrier which prevents speculative loads from bypassing earlier stores to the same virtual address under certain conditions.

The semantics of the Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the SSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order before the SSBB.
- When a load to a location appears in program order before the SSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store uses the same virtual address as the load.
  - The store appears in program order after the SSBB.

This is an alias of DSB. This means:

- The encodings in this description are named to match the encodings of DSB.
- The description of DSB gives the operational pseudocode for this instruction.



SSBB

is equivalent to

DSB #0

and is always the preferred disassembly.

Operation

The description of DSB gives the operational pseudocode for this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST2G

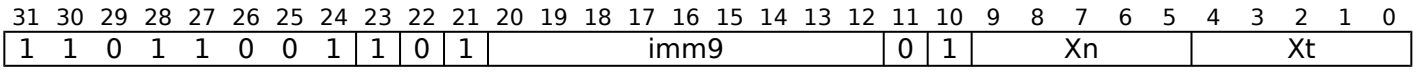
Store Allocation Tags stores an Allocation Tag to two Tag granules of memory. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: Post-index , Pre-index and Signed offset

Post-index

(FEAT_MTE)

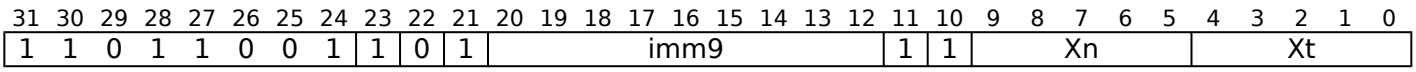


ST2G <Xt|SP>, [<Xn|SP>], #<simm>

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;
```

Pre-index

(FEAT_MTE)

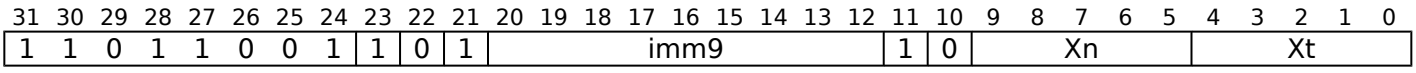


ST2G <Xt|SP>, [<Xn|SP>, #<simm>]!

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;
```

Signed offset

(FEAT_MTE)



ST2G <Xt|SP>, [<Xn|SP>{, #<simm>}]

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;
```

Assembler Symbols

<Xt|SP>	Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Xt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
<simm>	Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

Operation

```
bits(64) address;
bits(64) data = if t == 31 then SP[] else X[t, 64];
bits(4) tag = AArch64.AllocationTagFromAddress(data);

SetTagCheckedInstruction(FALSE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

AArch64.MemTag[address, AccType_NORMAL] = tag;
AArch64.MemTag[address+TAG_GRANULE, AccType_NORMAL] = tag;

if writeback then
    if postindex then
        address = address + offset;

    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

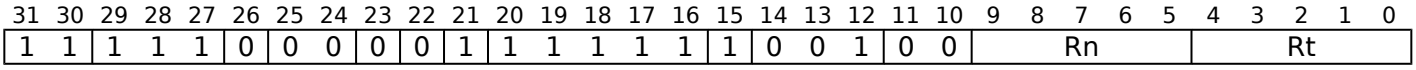
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST64B

Single-copy Atomic 64-byte Store without Return stores eight 64-bit doublewords from consecutive registers, Xt to X(t+7), to a memory location. The data that is stored is atomic and is required to be 64-byte-aligned.

Integer

(FEAT_LS64)



ST64B <Xt>, [<Xn|SP> {,#0}]

```
if !HaveFeatLS64() then UNDEFINED;
if Rt<4:3> == '11' || Rt<0> == '1' then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
boolean tag_checked = n != 31;
```

Assembler Symbols

<Xt>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckLDST64BEnabled();

bits(512) data;
bits(64) address;
bits(64) value;
acctype = AccType_ATOMICLS64;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

for i = 0 to 7
    value = X[t+i, 64];
    if BigEndian(acctype) then value = BigEndianReverse(value);
    data<63+64*i:64*i> = value;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

MemStore64B(address, data, acctype);
```

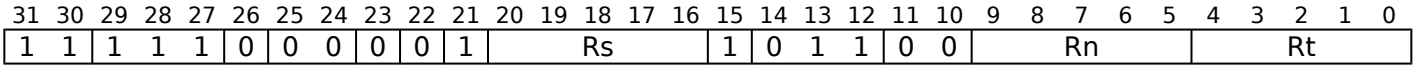
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST64BV

Single-copy Atomic 64-byte Store with Return stores eight 64-bit doublewords from consecutive registers, Xt to X(t+7), to a memory location, and writes the status result of the store to a register. The data that is stored is atomic and is required to be 64-byte aligned.

Integer

(FEAT_LS64_V)



ST64BV <Xs>, <Xt>, [<Xn|SP>]

```
if !HaveFeatLS64_V() then UNDEFINED;
if Rt<4:3> == '11' || Rt<0> == '1' then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);
boolean tag_checked = n != 31;
```

Assembler Symbols

<Xs> Is the 64-bit name of the general-purpose register into which the status result of this instruction is written, encoded in the "Rs" field.

The value returned is:

0xFFFFFFFF_FFFFFFFF
: If the memory location accessed does not support this instruction. In this case, the value at the memory location is UNKNOWN.

!= 0xFFFFFFFF_FFFFFFFF
: If the memory location accessed does support this instruction. In this case, the peripheral that provides the response defines the returned value and provides information on the state of the memory update at the memory location.

If XZR is used, then the return value is ignored.

<Xt> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckST64BVEnabled();

bits(512) data;
bits(64) address;
bits(64) value;
bits(64) status;
acctype = AccType_ATOMICLS64;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

for i = 0 to 7
    value = X[t+i, 64];
    if BigEndian(acctype) then value = BigEndianReverse(value);
    data<63+64*i:64*i> = value;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

status = MemStore64BWithRet(address, data, acctype);

if s != 31 then X[s, 64] = status;
```

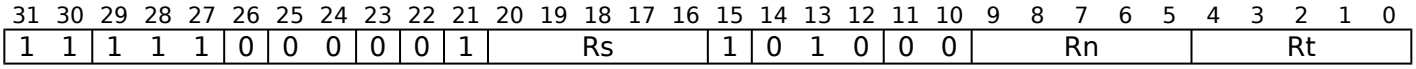
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST64BV0

Single-copy Atomic 64-byte EL0 Store with Return stores eight 64-bit doublewords from consecutive registers, Xt to X(t+7), to a memory location, with the bottom 32 bits taken from *ACCDATA_EL1*, and writes the status result of the store to a register. The data that is stored is atomic and is required to be 64-byte aligned.

Integer

(FEAT_LS64_ACCDATA)



ST64BV0 <Xs>, <Xt>, [<Xn|SP>]

```
if !HaveFeatLS64_ACCDATA() then UNDEFINED;
if Rt<4:3> == '11' || Rt<0> == '1' then UNDEFINED;

integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);
boolean tag_checked = n != 31;
```

Assembler Symbols

<Xs> Is the 64-bit name of the general-purpose register into which the status result of this instruction is written, encoded in the "Rs" field.

The value returned is:

0xFFFFFFFF_FFFFFFFF
: If the memory location accessed does not support this instruction. In this case, the value at the memory location is UNKNOWN.

!= 0xFFFFFFFF_FFFFFFFF
: If the memory location accessed does support this instruction. In this case, the peripheral that provides the response defines the returned value and provides information on the state of the memory update at the memory location.

If XZR is used, then the return value is ignored.

<Xt> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckST64BV0Enabled();

bits(512) data;
bits(64) address;
bits(64) value;
bits(64) status;
acctype = AccType_ATOMICLS64;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

bits(64) Xt = X[t, 64];
value<31:0> = ACCDATA_EL1<31:0>;
value<63:32> = Xt<63:32>;
if BigEndian(acctype) then value = BigEndianReverse(value);
data<63:0> = value;
for i = 1 to 7
    value = X[t+i, 64];
    if BigEndian(acctype) then value = BigEndianReverse(value);
    data<63+64*i:64*i> = value;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

status = MemStore64BWithRet(address, data, acctype);

if s != 31 then X[s, 64] = status;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STADD, STADDL

Atomic add on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory.

- `STADD` does not have release semantics.
- `STADDL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

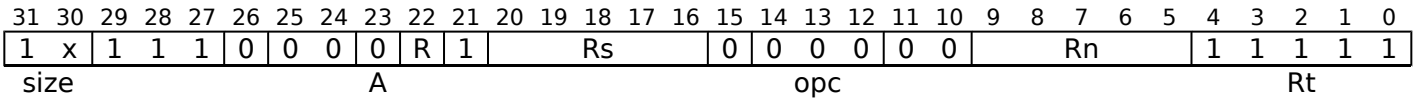
For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDADD, LDADDA, LDADDAL, LDADDL. This means:

- The encodings in this description are named to match the encodings of LDADD, LDADDA, LDADDAL, LDADDL.
- The description of LDADD, LDADDA, LDADDAL, LDADDL gives the operational pseudocode for this instruction.

Integer

(FEAT_LSE)



32-bit LDADD alias (size == 10 && R == 0)

```
STADD <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADD <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

32-bit LDADDL alias (size == 10 && R == 1)

```
STADDL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADDL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDADD alias (size == 11 && R == 0)

```
STADD <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDADD <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDADDL alias (size == 11 && R == 1)

```
STADDL <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDADDL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDADD, LDADDA, LDADDAL, LDADDL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STADDB, STADDLB

Atomic add on byte in memory, without return, atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory.

- `STADDB` does not have release semantics.
- `STADDLB` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

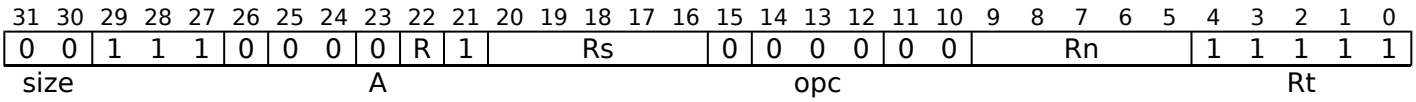
For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDADDB, LDADDAB, LDADDALB, LDADDLB. This means:

- The encodings in this description are named to match the encodings of LDADDB, LDADDAB, LDADDALB, LDADDLB.
- The description of LDADDB, LDADDAB, LDADDALB, LDADDLB gives the operational pseudocode for this instruction.

Integer

(FEAT_LSE)



No memory ordering (R == 0)

```
STADDB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADDB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release (R == 1)

```
STADDLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADDLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDADDB, LDADDAB, LDADDALB, LDADDLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STADDH, STADDLH

Atomic add on halfword in memory, without return, atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory.

- STADDH does not have release semantics.
- STADDLH stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

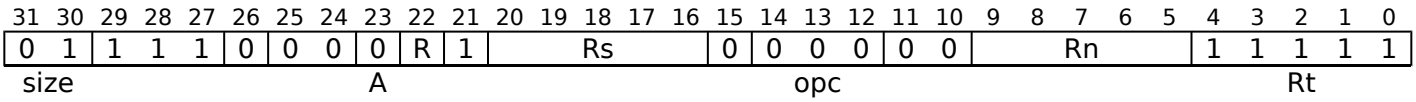
For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDADDH, LDADDAH, LDADDALH, LDADDLH. This means:

- The encodings in this description are named to match the encodings of LDADDH, LDADDAH, LDADDALH, LDADDLH.
- The description of LDADDH, LDADDAH, LDADDALH, LDADDLH gives the operational pseudocode for this instruction.

Integer

(FEAT_LSE)



No memory ordering (R == 0)

```
STADDH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADDH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release (R == 1)

```
STADDLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDADDLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDADDH, LDADDAH, LDADDALH, LDADDLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STCLR, STCLRL

Atomic bit clear on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- STCLR does not have release semantics.
- STCLRL stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

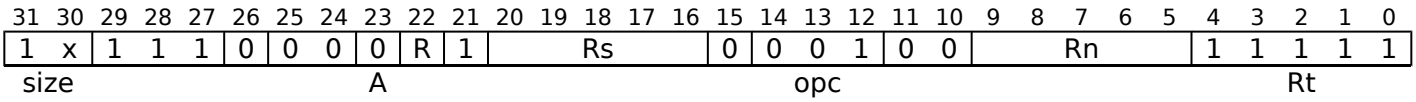
For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDCLR, LDCLRA, LDCLRAL, LDCLRL. This means:

- The encodings in this description are named to match the encodings of LDCLR, LDCLRA, LDCLRAL, LDCLRL.
- The description of LDCLR, LDCLRA, LDCLRAL, LDCLRL gives the operational pseudocode for this instruction.

Integer

(FEAT_LSE)



32-bit LDCLR alias (size == 10 && R == 0)

```
STCLR <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLR <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

32-bit LDCLRL alias (size == 10 && R == 1)

```
STCLRL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLRL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDCLR alias (size == 11 && R == 0)

```
STCLR <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDCLR <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDCLRL alias (size == 11 && R == 1)

```
STCLRL <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDCLRL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDCLR, LDCLRA, LDCLRAL, LDCLRL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STCLRB, STCLRLB

Atomic bit clear on byte in memory, without return, atomically loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- STCLRB does not have release semantics.
- STCLRLB stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

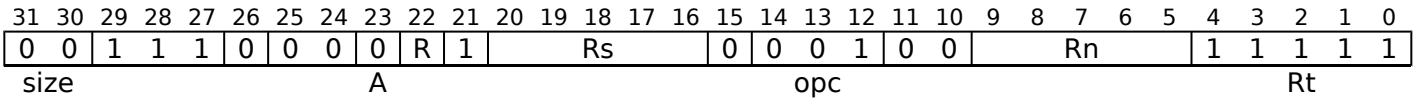
For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB. This means:

- The encodings in this description are named to match the encodings of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB.
- The description of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB gives the operational pseudocode for this instruction.

Integer

(FEAT_LSE)



No memory ordering (R == 0)

```
STCLRB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLRB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release (R == 1)

```
STCLRLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLRLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STCLRH, STCLRLH

Atomic bit clear on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- STCLRH does not have release semantics.
- STCLRLH stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

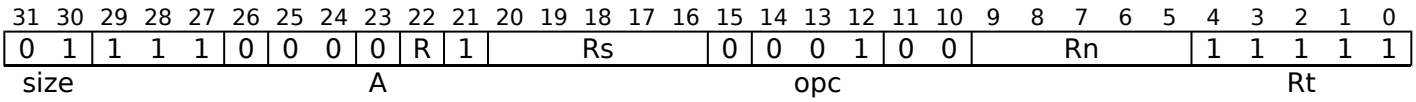
For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH. This means:

- The encodings in this description are named to match the encodings of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH.
- The description of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH gives the operational pseudocode for this instruction.

Integer

(FEAT_LSE)



No memory ordering (R == 0)

```
STCLRH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLRH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release (R == 1)

```
STCLRLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDCLRLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STEOR, STEORL

Atomic exclusive OR on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- `STEOR` does not have release semantics.
- `STEORL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

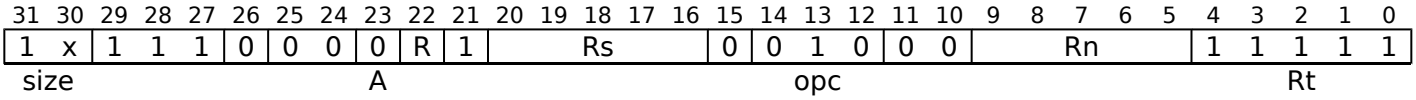
For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDEOR, LDEORA, LDEORAL, LDEORL. This means:

- The encodings in this description are named to match the encodings of LDEOR, LDEORA, LDEORAL, LDEORL.
- The description of LDEOR, LDEORA, LDEORAL, LDEORL gives the operational pseudocode for this instruction.

Integer

(FEAT_LSE)



32-bit LDEOR alias (size == 10 && R == 0)

```
STEOR <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEOR <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

32-bit LDEORL alias (size == 10 && R == 1)

```
STEORL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEORL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDEOR alias (size == 11 && R == 0)

```
STEOR <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDEOR <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDEORL alias (size == 11 && R == 1)

```
STEORL <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDEORL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDEOR, LDEORA, LDEORAL, LDEORL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STEORB, STEORLB

Atomic exclusive OR on byte in memory, without return, atomically loads an 8-bit byte from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- `STEORB` does not have release semantics.
- `STEORLB` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

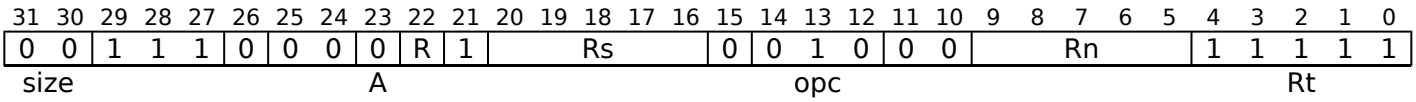
For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDEORB, LDEORAB, LDEORALB, LDEORLB. This means:

- The encodings in this description are named to match the encodings of LDEORB, LDEORAB, LDEORALB, LDEORLB.
- The description of LDEORB, LDEORAB, LDEORALB, LDEORLB gives the operational pseudocode for this instruction.

Integer

(FEAT_LSE)



No memory ordering (R == 0)

```
STEORB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEORB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release (R == 1)

```
STEORLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEORLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDEORB, LDEORAB, LDEORALB, LDEORLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STEORH, STEORLH

Atomic exclusive OR on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- `STEORH` does not have release semantics.
- `STEORLH` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

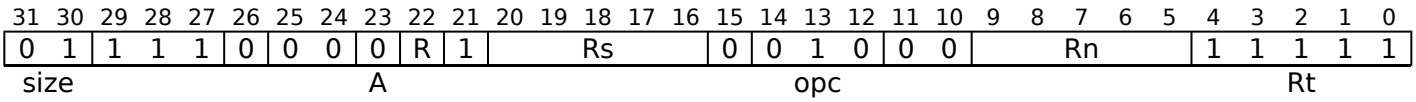
For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDEORH, LDEORAH, LDEORALH, LDEORLH. This means:

- The encodings in this description are named to match the encodings of LDEORH, LDEORAH, LDEORALH, LDEORLH.
- The description of LDEORH, LDEORAH, LDEORALH, LDEORLH gives the operational pseudocode for this instruction.

Integer

(FEAT_LSE)



No memory ordering (R == 0)

```
STEORH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEORH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release (R == 1)

```
STEORLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDEORLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDEORH, LDEORAH, LDEORALH, LDEORLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STG

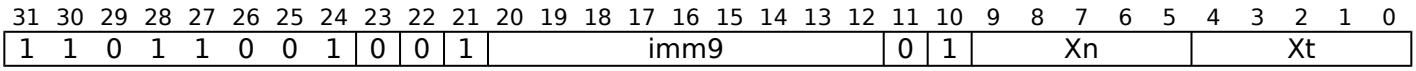
Store Allocation Tag stores an Allocation Tag to memory. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: Post-index , Pre-index and Signed offset

Post-index

(FEAT_MTE)

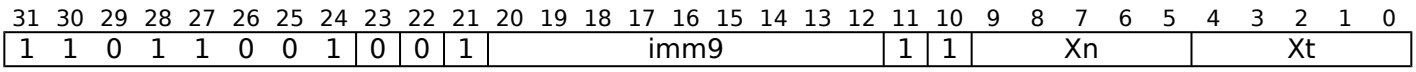


STG <Xt|SP>, [<Xn|SP>], #<simm>

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;
```

Pre-index

(FEAT_MTE)

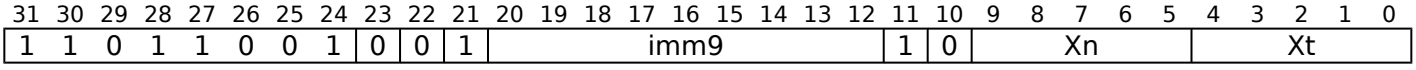


STG <Xt|SP>, [<Xn|SP>, #<simm>]!

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;
```

Signed offset

(FEAT_MTE)



STG <Xt|SP>, [<Xn|SP>{, #<simm>}]

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;
```

Assembler Symbols

<Xt|SP>	Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Xt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
<simm>	Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

Operation

```
bits(64) address;

SetTagCheckedInstruction(FALSE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

bits(64) data = if t == 31 then SP[] else X[t, 64];
bits(4) tag = AArch64.AllocationTagFromAddress(data);
AArch64.MemTag[address, AccType_NORMAL] = tag;

if writeback then
    if postindex then
        address = address + offset;

    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

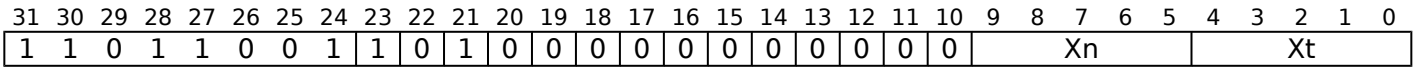
STGM

Store Tag Multiple writes a naturally aligned block of N Allocation Tags, where the size of N is identified in GMID_EL1.BS, and the Allocation Tag written to address A is taken from the source register at 4*A<7:4>+3:4*A<7:4>.

This instruction is UNDEFINED at EL0.

This instruction generates an Unchecked access.

Integer
(FEAT_MTE2)



STGM <Xt>, [<Xn|SP>]

```
if !HaveMTE2Ext() then UNDEFINED;
integer t = UInt(Xt);
integer n = UInt(Xn);
```

Assembler Symbols

<Xt>	Is the 64-bit name of the general-purpose source register, encoded in the "Xt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

Operation

```
if PSTATE.EL == EL0 then
    UNDEFINED;

bits(64) data = X[t, 64];
bits(64) address;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

integer size = 4 * (2 ^ (UInt(GMID_EL1.BS)));
address = Align(address, size);
integer count = size >> LOG2_TAG_GRANULE;
integer index = UInt(address<LOG2_TAG_GRANULE+3:LOG2_TAG_GRANULE>);

for i = 0 to count-1
    bits(4) tag = data<(index*4)+3:index*4>;
    AArch64.MemTag[address, AccType_NORMAL] = tag;
    address = address + TAG_GRANULE;
    index = index + 1;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STGP

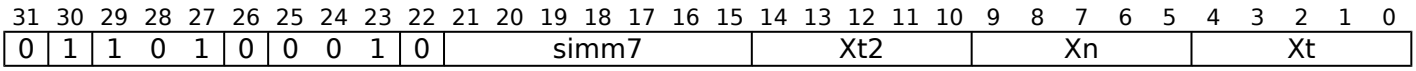
Store Allocation Tag and Pair of registers stores an Allocation Tag and two 64-bit doublewords to memory, from two registers. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the base register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: Post-index , Pre-index and Signed offset

Post-index

(FEAT_MTE)

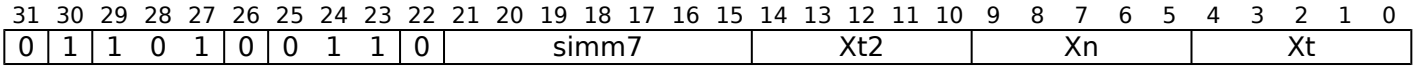


STGP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
integer t2 = UInt(Xt2);
bits(64) offset = LSL(SignExtend(simm7, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;
```

Pre-index

(FEAT_MTE)

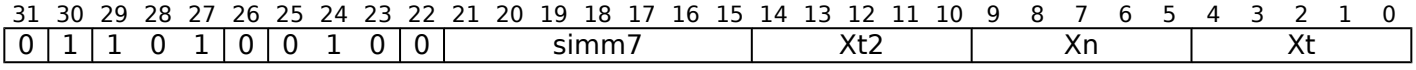


STGP <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
integer t2 = UInt(Xt2);
bits(64) offset = LSL(SignExtend(simm7, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;
```

Signed offset

(FEAT_MTE)



STGP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
integer t2 = UInt(Xt2);
bits(64) offset = LSL(SignExtend(simm7, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;
```

Assembler Symbols

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Xt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Xt2" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

<imm> For the post-index and pre-index variant: is the signed immediate offset, a multiple of 16 in the range -1024 to 1008, encoded in the "simm7" field.

For the signed offset variant: is the optional signed immediate offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "simm7" field.

Operation

```
bits(64) address;
bits(64) data1;
bits(64) data2;

SetTagCheckedInstruction(FALSE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data1 = X[t, 64];
data2 = X[t2, 64];

if !postindex then
    address = address + offset;

if address != Align(address, TAG_GRANULE) then
    AArch64.Abort(address, AlignmentFault(AccType_NORMAL, TRUE, FALSE));

Mem[address, 8, AccType_NORMAL] = data1;
Mem[address+8, 8, AccType_NORMAL] = data2;

AArch64.MemTag[address, AccType_NORMAL] = AArch64.AllocationTagFromAddress(address);

if writeback then
    if postindex then
        address = address + offset;

    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

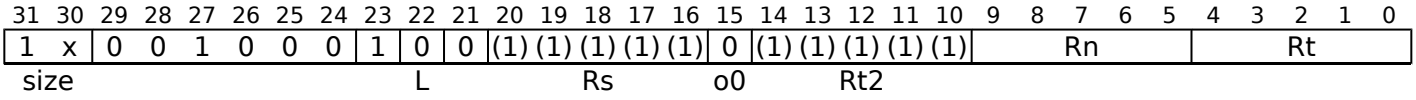
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STLLR

Store LORelease Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

No offset

(FEAT_LOR)



32-bit (size == 10)

```
STLLR <Wt>, [<Xn|SP>{,#0}]
```

64-bit (size == 11)

```
STLLR <Xt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
boolean tag_checked = n != 31;
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = X[t, elsize];
Mem[address, dbytes, AccType_LIMITEDORDERED] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

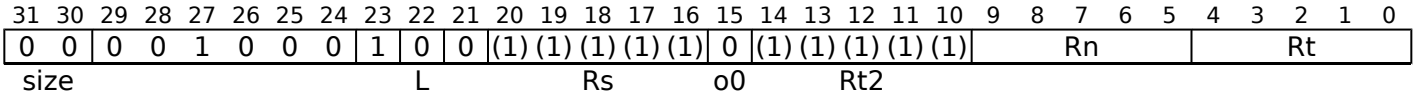
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STLLRB

Store LORelease Register Byte stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

No offset

(FEAT_LOR)




```
STLLRB <Wt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = X[t, 8];
Mem[address, 1, AccType_LIMITEDORDERED] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

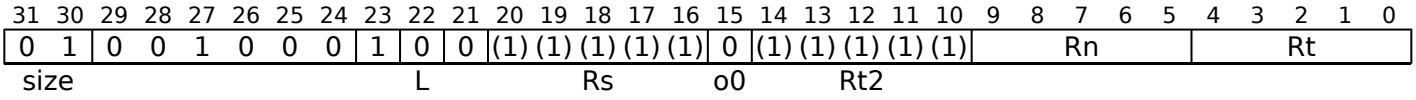
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STLLRH

Store LORelease Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

No offset

(FEAT_LOR)




```
STLLRH <Wt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = X[t, 16];
Mem[address, 2, AccType_LIMITEDORDERED] = data;
```

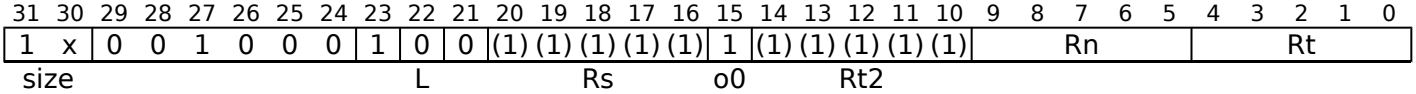
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STLR

Store-Release Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.



32-bit (size == 10)

```
STLR <Wt>, [<Xn|SP>{,#0}]
```

64-bit (size == 11)

```
STLR <Xt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer elsize = 8 << UInt(size);
boolean tag_checked = n != 31;
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = X[t, elsize];
Mem[address, dbytes, AccType_ORDERED] = data;
```

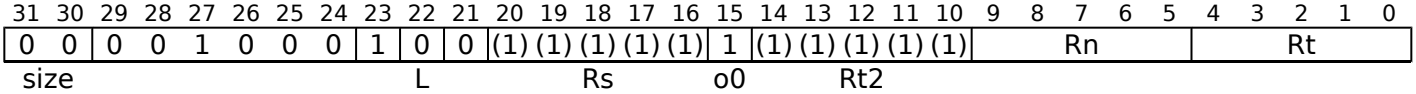
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STLRB

Store-Release Register Byte stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.



STLRB <Wt>, [<Xn|SP>{,#0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = X[t, 8];
Mem[address, 1, AccType_ORDERED] = data;
```

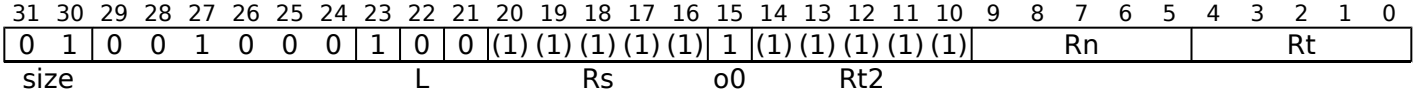
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STLRH

Store-Release Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.



STLRH <Wt>, [<Xn|SP>{,#0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

data = X[t, 16];
Mem[address, 2, AccType_ORDERED] = data;
```

Operational information

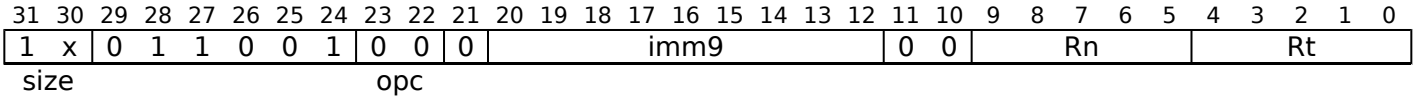
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STLUR

Store-Release Register (unscaled) calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register.
The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*
For information about memory accesses, see *Load/Store addressing modes*.

Unscaled offset
(FEAT_LRCPC2)



32-bit (size == 10)

```
STLUR <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit (size == 11)

```
STLUR <Xt>, [<Xn|SP>{, #<simm>}]
```

```
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer datasize = 8 << scale;
boolean tag_checked = n != 31;
```

Operation

```
bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = X[t, datasize];
Mem[address, datasize DIV 8, AccType_ORDERED] = data;
```

Operational information

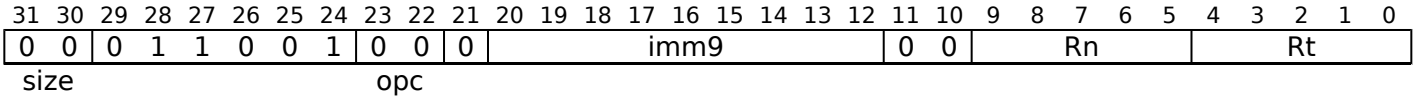
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STLURB

Store-Release Register Byte (unscaled) calculates an address from a base register value and an immediate offset, and stores a byte to the calculated address, from a 32-bit register.
The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*
For information about memory accesses, see *Load/Store addressing modes*.

Unscaled offset
(FEAT_LRCPC2)




```
STLURB <Wt>, [<Xn|SP>{, #<simm>}]
```

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Operation

```
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = X[t, 8];
Mem[address, 1, AccType_ORDERED] = data;
```

Operational information

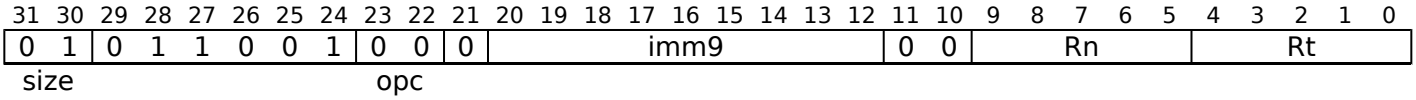
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STLURH

Store-Release Register Halfword (unscaled) calculates an address from a base register value and an immediate offset, and stores a halfword to the calculated address, from a 32-bit register.
The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*
For information about memory accesses, see *Load/Store addressing modes*.

Unscaled offset
(FEAT_LRCPC2)




```
STLURH <Wt>, [<Xn|SP>{, #<simm>}]

bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Operation

```
bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = X[t, 16];
Mem[address, 2, AccType_ORDERED] = data;
```

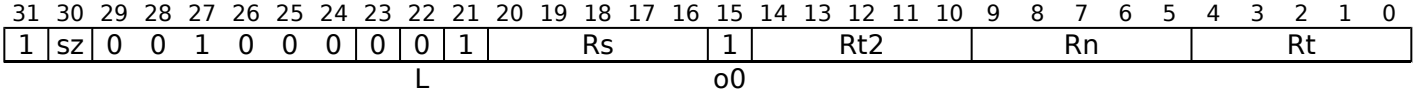
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STLXP

Store-Release Exclusive Pair of registers stores two 32-bit words or two 64-bit doublewords to a memory location if the PE has exclusive access to the memory address, from two registers, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. For information on single-copy atomicity and alignment requirements, see *Requirements for single-copy atomicity* and *Alignment of data accesses*. If a 64-bit pair Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. The instruction also has memory ordering semantics, as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.



32-bit (sz == 0)

```
STLXP <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{,#0}]
```

64-bit (sz == 1)

```
STLXP <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);    // ignored by load/store single register
integer s = UInt(Rs);    // ignored by all loads and store-release

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t || (s == t2) then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STLXP*.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0
If the operation updates memory.

1
If the operation fails to update memory.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    bits(datasize DIV 2) el1 = X[t, datasize DIV 2];
    bits(datasize DIV 2) el2 = X[t2, datasize DIV 2];
    data = if BigEndian(AccType_ORDEREDATOMIC) then el1:el2 else el2:el1;
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, dbytes) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, AccType_ORDEREDATOMIC] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

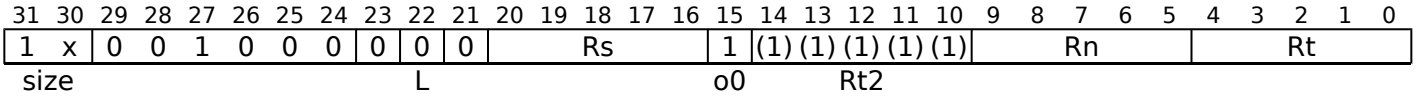
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STLXR

Store-Release Exclusive Register stores a 32-bit word or a 64-bit doubleword to memory if the PE has exclusive access to the memory address, from two registers, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. The memory access is atomic. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.



32-bit (size == 10)

```
STLXR <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

64-bit (size == 11)

```
STLXR <Ws>, <Xt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release

integer elsize = 8 << UInt(size);
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STLXR*.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0
If the operation updates memory.

1
If the operation fails to update memory.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.

- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(elsize) UNKNOWN;
else
    data = X[t, elsize];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, dbytes) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, AccType_ORDEREDATOMIC] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

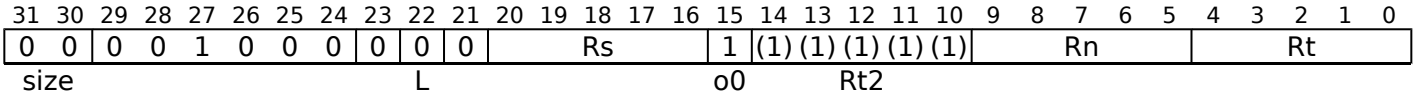
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STLXRB

Store-Release Exclusive Register Byte stores a byte from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. The memory access is atomic. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.



STLXRB <Ws>, <Wt>, [<Xn|SP>{,#0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release

boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STLXRB*.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0
If the operation updates memory.

1
If the operation fails to update memory.

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = X[t, 8];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 1) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 1, AccType_ORDEREDATOMIC] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

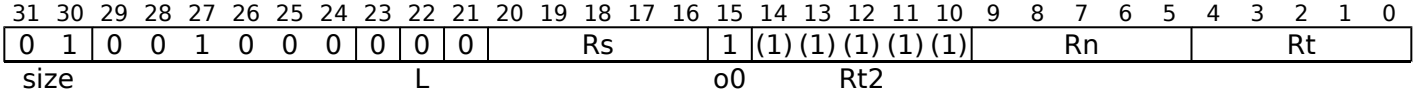
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STLXRH

Store-Release Exclusive Register Halfword stores a halfword from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. The memory access is atomic. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses see *Load/Store addressing modes*.



STLXRH <Ws>, <Wt>, [<Xn|SP>{,#0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release

boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STLXRH*.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0
If the operation updates memory.

1
If the operation fails to update memory.

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t, 16];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 2) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 2, AccType_ORDEREDATOMIC] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

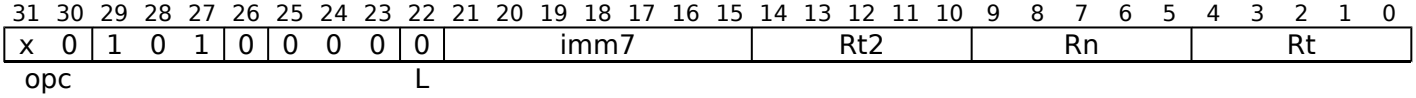
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STNP

Store Pair of Registers, with non-temporal hint, calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about memory accesses, see *Load/Store addressing modes*. For information about Non-temporal pair instructions, see *Load/Store Non-temporal pair*.



32-bit (opc == 00)

```
STNP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

64-bit (opc == 10)

```
STNP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

```
// Empty.
```

Assembler Symbols

<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4. For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc<0> == '1' then UNDEFINED;
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = n != 31;
```

Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data1 = X[t, datasize];
data2 = X[t2, datasize];
Mem[address, dbytes, AccType_STREAM] = data1;
Mem[address+dbytes, dbytes, AccType_STREAM] = data2;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

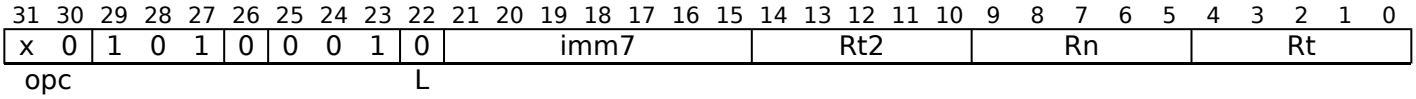
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STP

Store Pair of Registers calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Signed offset

Post-index



32-bit (opc == 00)

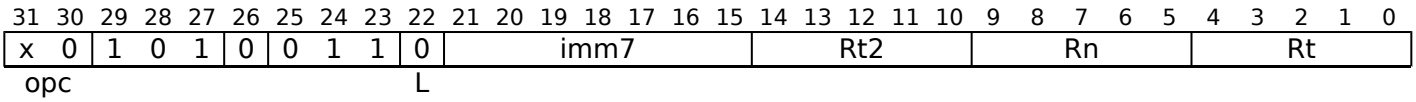
```
STP <Wt1>, <Wt2>, [<Xn|SP>], #<imm>
```

64-bit (opc == 10)

```
STP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

Pre-index



32-bit (opc == 00)

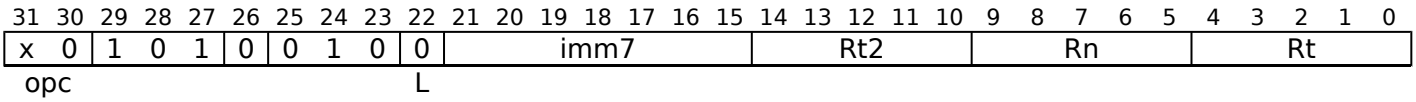
```
STP <Wt1>, <Wt2>, [<Xn|SP>, #<imm>]!
```

64-bit (opc == 10)

```
STP <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

Signed offset



32-bit (opc == 00)

```
STP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}]
```

64-bit (opc == 10)

```
STP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STP*.

Assembler Symbols

<Wt1>	Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Wt2>	Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xt1>	Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.
<Xt2>	Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4. For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4. For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8. For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if L:opc<0> == '01' || opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;

if wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is pre-writeback
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Operation

```
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

if rt_unknown && t == n then
    data1 = bits(datasize) UNKNOWN;
else
    data1 = X[t, datasize];
if rt_unknown && t2 == n then
    data2 = bits(datasize) UNKNOWN;
else
    data2 = X[t2, datasize];
if HaveLSE2Ext() then
    bits(2*datasize) full_data;
    if BigEndian(AccType_NORMAL) then
        full_data = data1:data2;
    else
        full_data = data2:data1;
    Mem[address, 2*dbytes, AccType_NORMAL, TRUE] = full_data;
else
    Mem[address, dbytes, AccType_NORMAL] = data1;
    Mem[address+dbytes, dbytes, AccType_NORMAL] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

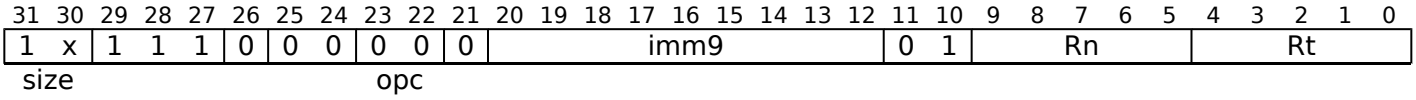
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STR (immediate)

Store Register (immediate) stores a word or a doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index



32-bit (size == 10)

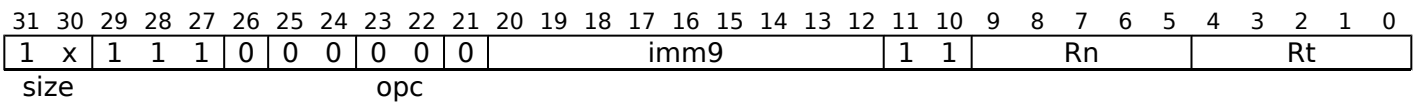
```
STR <Wt>, [<Xn|SP>], #<simm>
```

64-bit (size == 11)

```
STR <Xt>, [<Xn|SP>], #<simm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index



32-bit (size == 10)

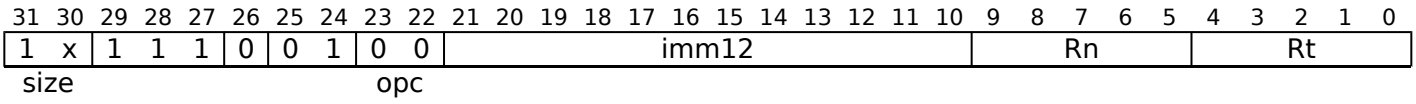
```
STR <Wt>, [<Xn|SP>, #<simm>]!
```

64-bit (size == 11)

```
STR <Xt>, [<Xn|SP>, #<simm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



32-bit (size == 10)

```
STR <Wt>, [<Xn|SP>{, #<pimm>}]
```

64-bit (size == 11)

```
STR <Xt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4. For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer datasize = 8 << scale;
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;
Constraint c;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Operation

```
bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    data = X[t, datasize];
Mem[address, datasize DIV 8, AccType_NORMAL] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

Operational information

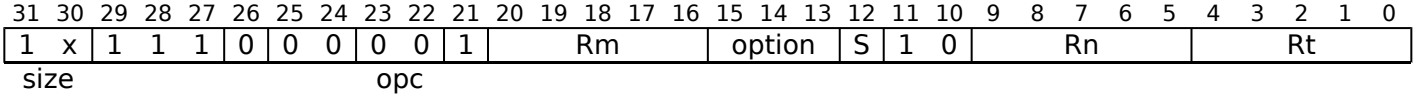
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STR (register)

Store Register (register) calculates an address from a base register value and an offset register value, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about memory accesses, see *Load/Store addressing modes*.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.



32-bit (size == 10)

```
STR <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

64-bit (size == 11)

```
STR <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount> For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#2

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#3

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);

integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = X[t, datasize];
Mem[address, datasize DIV 8, AccType_NORMAL] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

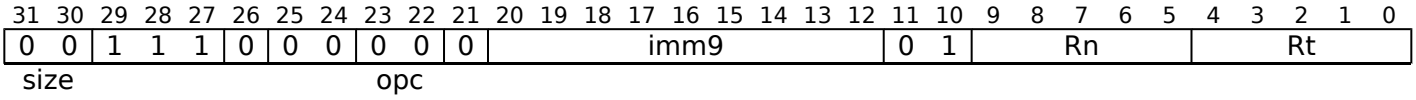
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STRB (immediate)

Store Register Byte (immediate) stores the least significant byte of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

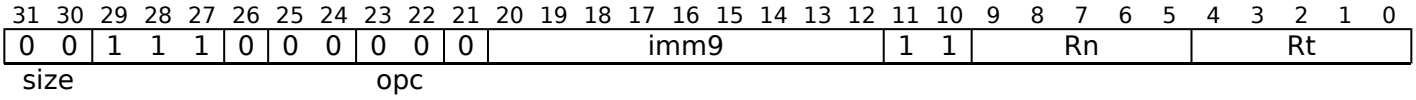
Post-index



STRB <Wt>, [<Xn|SP>], #<simm>

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

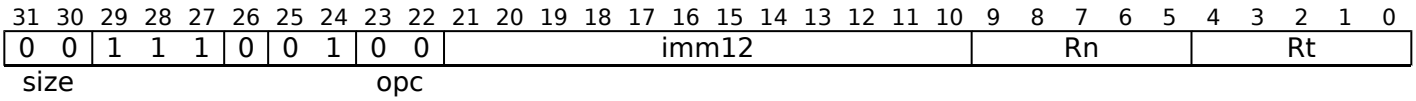
Pre-index



STRB <Wt>, [<Xn|SP>, #<simm>]!

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



STRB <Wt>, [<Xn|SP>{, #<pimm>}]

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 0);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STRB (immediate)*.

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;
Constraint c;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Operation

```
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = X[t, 8];
Mem[address, 1, AccType_NORMAL] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

Operational information

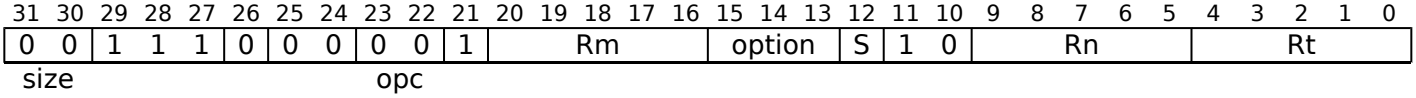
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STRB (register)

Store Register Byte (register) calculates an address from a base register value and an offset register value, and stores a byte from a 32-bit register to the calculated address. For information about memory accesses, see *Load/Store addressing modes*.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.



Extended register (option != 011)

```
STRB <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

Shifted register (option == 011)

```
STRB <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

```
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Wm>	When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
<Xm>	When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.
<extend>	Is the index extend specifier, encoded in “option”:

option	<extend>
010	UXTW
110	SXTW
111	SXTX

<amount>	Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, 0, 64);
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = X[t, 8];
Mem[address, 1, AccType_NORMAL] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

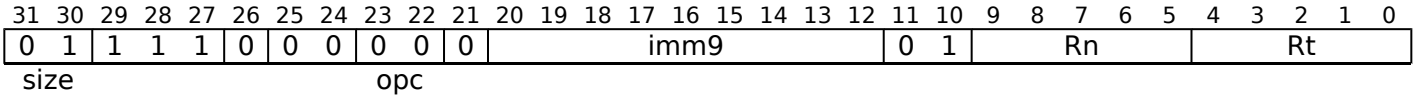
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STRH (immediate)

Store Register Halfword (immediate) stores the least significant halfword of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

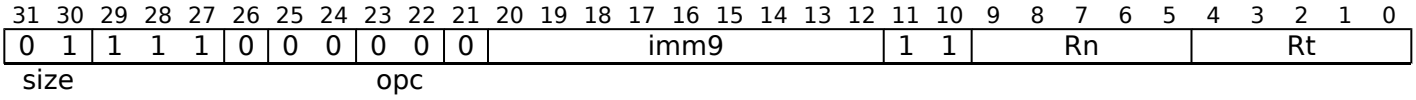
Post-index



STRH <Wt>, [<Xn|SP>], #<simm>

```
boolean wback = TRUE;
boolean postindex = TRUE;
bits(64) offset = SignExtend(imm9, 64);
```

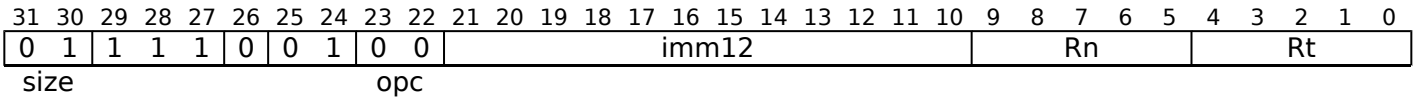
Pre-index



STRH <Wt>, [<Xn|SP>, #<simm>]!

```
boolean wback = TRUE;
boolean postindex = FALSE;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



STRH <Wt>, [<Xn|SP>{, #<pimm>}]

```
boolean wback = FALSE;
boolean postindex = FALSE;
bits(64) offset = LSL(ZeroExtend(imm12, 64), 1);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STRH (immediate)*.

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;
Constraint c;

if wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE    rt_unknown = FALSE;    // value stored is original value
        when Constraint_UNKNOWN rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Operation

```
bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t, 16];
Mem[address, 2, AccType_NORMAL] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

Operational information

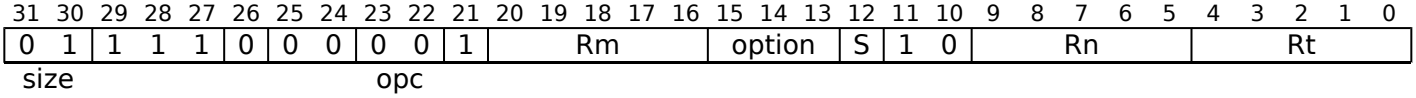
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STRH (register)

Store Register Halfword (register) calculates an address from a base register value and an offset register value, and stores a halfword from a 32-bit register to the calculated address. For information about memory accesses, see *Load/Store addressing modes*.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.




```
STRH <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then 1 else 0;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#1

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift, 64);
bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(TRUE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = X[t, 16];
Mem[address, 2, AccType_NORMAL] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STSET, STSETL

Atomic bit set on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- STSET does not have release semantics.
- STSETL stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

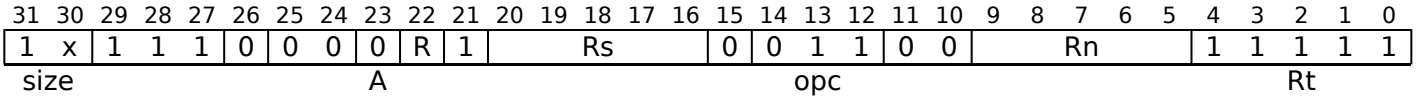
For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDSET, LDSETA, LDSETAL, LDSETL. This means:

- The encodings in this description are named to match the encodings of LDSET, LDSETA, LDSETAL, LDSETL.
- The description of LDSET, LDSETA, LDSETAL, LDSETL gives the operational pseudocode for this instruction.

Integer

(FEAT_LSE)



32-bit LDSET alias (size == 10 && R == 0)

```
STSET <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSET <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

32-bit LDSETL alias (size == 10 && R == 1)

```
STSETL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSETL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDSET alias (size == 11 && R == 0)

```
STSET <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSET <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDSETL alias (size == 11 && R == 1)

```
STSETL <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSETL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSET, LDSETA, LDSETAL, LDSETL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STSETB, STSETLB

Atomic bit set on byte in memory, without return, atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- STSETB does not have release semantics.
- STSETLB stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

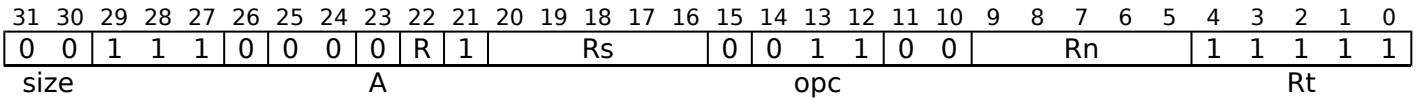
For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDSETB, LDSETAB, LDSETALB, LDSETLB. This means:

- The encodings in this description are named to match the encodings of LDSETB, LDSETAB, LDSETALB, LDSETLB.
- The description of LDSETB, LDSETAB, LDSETALB, LDSETLB gives the operational pseudocode for this instruction.

Integer

(FEAT_LSE)



No memory ordering (R == 0)

```
STSETB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSETB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release (R == 1)

```
STSETLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSETLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSETB, LDSETAB, LDSETALB, LDSETLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STSETH, STSETLH

Atomic bit set on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- STSETH does not have release semantics.
- STSETLH stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

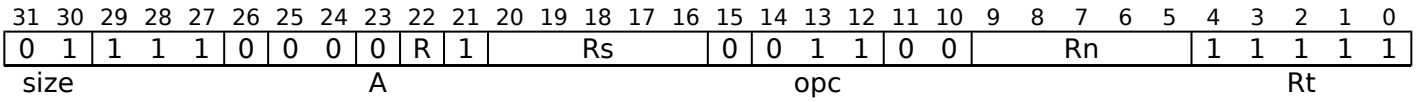
For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDSETH, LDSETAH, LDSETALH, LDSETLH. This means:

- The encodings in this description are named to match the encodings of LDSETH, LDSETAH, LDSETALH, LDSETLH.
- The description of LDSETH, LDSETAH, LDSETALH, LDSETLH gives the operational pseudocode for this instruction.

Integer

(FEAT_LSE)



No memory ordering (R == 0)

```
STSETH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSETH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release (R == 1)

```
STSETLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSETLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSETH, LDSETAH, LDSETALH, LDSETLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STSMAX, STSMAXL

Atomic signed maximum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- STSMAX does not have release semantics.
- STSMAXL stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

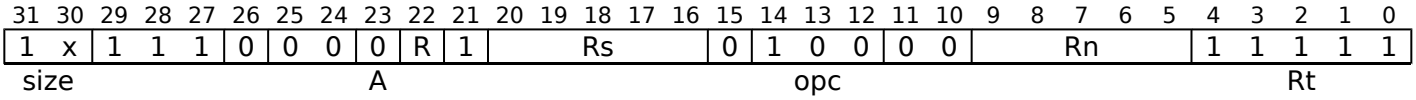
For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL. This means:

- The encodings in this description are named to match the encodings of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL.
- The description of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL gives the operational pseudocode for this instruction.

Integer

(FEAT_LSE)



32-bit LDSMAX alias (size == 10 && R == 0)

```
STSMAX <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAX <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

32-bit LDSMAXL alias (size == 10 && R == 1)

```
STSMAXL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDSMAX alias (size == 11 && R == 0)

```
STSMAX <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSMAX <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDSMAXL alias (size == 11 && R == 1)

```
STSMAXL <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STSMAXB, STSMAXLB

Atomic signed maximum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- `STSMAXB` does not have release semantics.
- `STSMAXLB` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

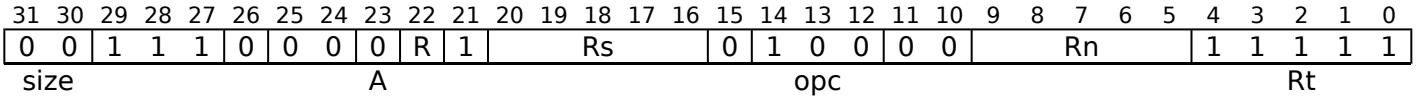
For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB. This means:

- The encodings in this description are named to match the encodings of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB.
- The description of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB gives the operational pseudocode for this instruction.

Integer

(FEAT_LSE)



No memory ordering (R == 0)

```
STSMAXB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release (R == 1)

```
STSMAXLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STSMAXH, STSMAXLH

Atomic signed maximum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- `STSMAXH` does not have release semantics.
- `STSMAXLH` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

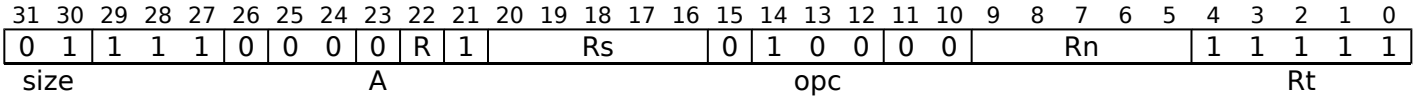
For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH. This means:

- The encodings in this description are named to match the encodings of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH.
- The description of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH gives the operational pseudocode for this instruction.

Integer

(FEAT_LSE)



No memory ordering (R == 0)

```
STSMAXH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release (R == 1)

```
STSMAXLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMAXLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STSMIN, STSMINL

Atomic signed minimum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- `STSMIN` does not have release semantics.
- `STSMINL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

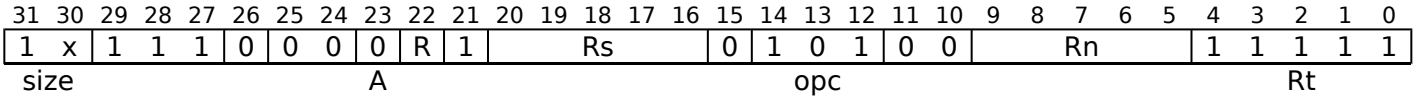
For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDSMIN, LDSMINA, LDSMINAL, LDSMINL. This means:

- The encodings in this description are named to match the encodings of LDSMIN, LDSMINA, LDSMINAL, LDSMINL.
- The description of LDSMIN, LDSMINA, LDSMINAL, LDSMINL gives the operational pseudocode for this instruction.

Integer

(FEAT_LSE)



32-bit LDSMIN alias (size == 10 && R == 0)

```
STSMIN <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMIN <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

32-bit LDSMINL alias (size == 10 && R == 1)

```
STSMINL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMINL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDSMIN alias (size == 11 && R == 0)

```
STSMIN <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSMIN <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDSMINL alias (size == 11 && R == 1)

```
STSMINL <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDSMINL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMIN, LDSMINA, LDSMINAL, LDSMINL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STSMINB, STSMINLB

Atomic signed minimum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- `STSMINB` does not have release semantics.
- `STSMINLB` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

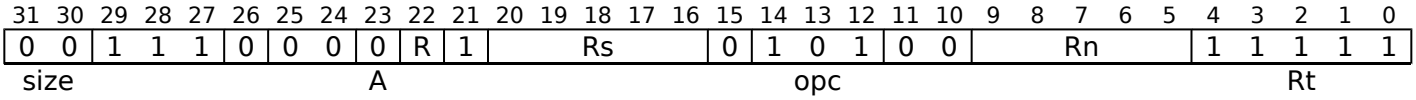
For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB. This means:

- The encodings in this description are named to match the encodings of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB.
- The description of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB gives the operational pseudocode for this instruction.

Integer

(FEAT_LSE)



No memory ordering (R == 0)

```
STSMINB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMINB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release (R == 1)

```
STSMINLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMINLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STSMINH, STSMINLH

Atomic signed minimum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- `STSMINH` does not have release semantics.
- `STSMINLH` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

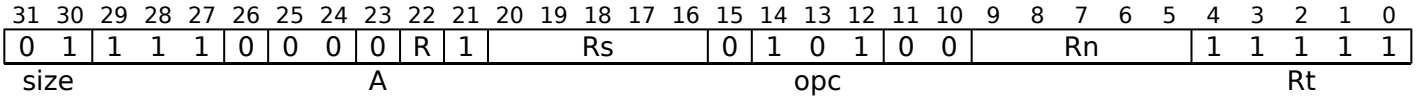
For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH. This means:

- The encodings in this description are named to match the encodings of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH.
- The description of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH gives the operational pseudocode for this instruction.

Integer

(FEAT_LSE)



No memory ordering (R == 0)

```
STSMINH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMINH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release (R == 1)

```
STSMINLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDSMINLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

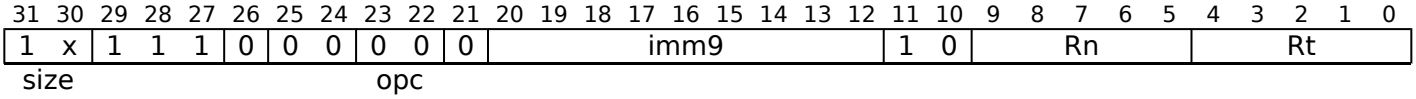
STTR

Store Register (unprivileged) stores a word or doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of *HCR_EL2*.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.



32-bit (size == 10)

```
STTR <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit (size == 11)

```
STTR <Xt>, [<Xn|SP>{, #<simm>}]
```

```
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype;
unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

integer datasize = 8 << scale;
boolean tag_checked = n != 31;
```

Operation

```
bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = X[t, datasize];
Mem[address, datasize DIV 8, acctype] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

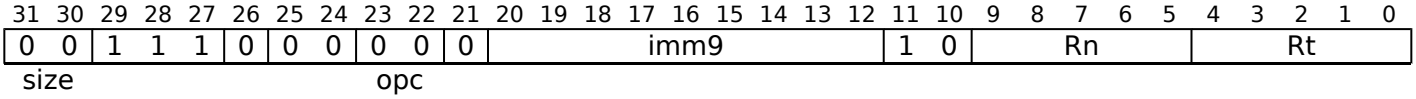
STTRB

Store Register Byte (unprivileged) stores a byte from a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of *HCR_EL2*.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.



STTRB <Wt>, [<Xn|SP>{, #<simm>}]

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype;
unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

boolean tag_checked = n != 31;
```

Operation

```
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = X[t, 8];
Mem[address, 1, acctype] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

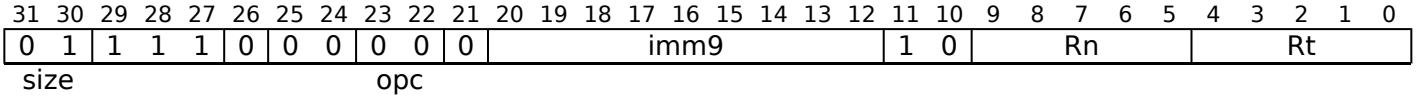
STTRH

Store Register Halfword (unprivileged) stores a halfword from a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of *HCR_EL2*.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.



STTRH <Wt>, [<Xn|SP>{, #<simm>}]

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype;
unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() && HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

boolean tag_checked = n != 31;
```

Operation

```
bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = X[t, 16];
Mem[address, 2, acctype] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STUMAX, STUMAXL

Atomic unsigned maximum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- `STUMAX` does not have release semantics.
- `STUMAXL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

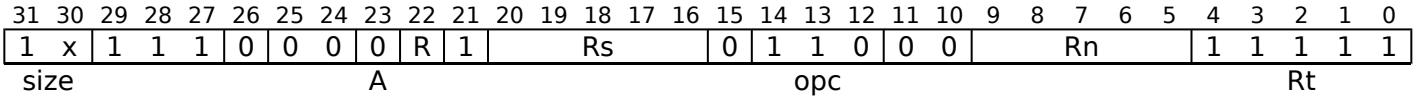
For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL. This means:

- The encodings in this description are named to match the encodings of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL.
- The description of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL gives the operational pseudocode for this instruction.

Integer

(FEAT_LSE)



32-bit LDUMAX alias (size == 10 && R == 0)

```
STUMAX <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAX <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

32-bit LDUMAXL alias (size == 10 && R == 1)

```
STUMAXL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDUMAX alias (size == 11 && R == 0)

```
STUMAX <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDUMAX <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDUMAXL alias (size == 11 && R == 1)

```
STUMAXL <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STUMAXB, STUMAXLB

Atomic unsigned maximum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- `STUMAXB` does not have release semantics.
- `STUMAXLB` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

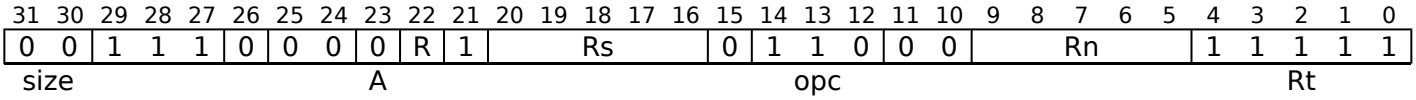
For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB. This means:

- The encodings in this description are named to match the encodings of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB.
- The description of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB gives the operational pseudocode for this instruction.

Integer

(FEAT_LSE)



No memory ordering (R == 0)

```
STUMAXB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release (R == 1)

```
STUMAXLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STUMAXH, STUMAXLH

Atomic unsigned maximum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- STUMAXH does not have release semantics.
- STUMAXLH stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

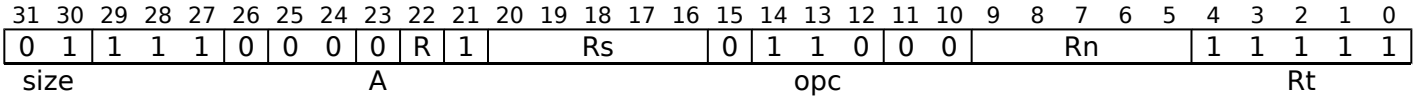
For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH. This means:

- The encodings in this description are named to match the encodings of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH.
- The description of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH gives the operational pseudocode for this instruction.

Integer

(FEAT_LSE)



No memory ordering (R == 0)

```
STUMAXH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release (R == 1)

```
STUMAXLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMAXLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STUMIN, STUMINL

Atomic unsigned minimum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- `STUMIN` does not have release semantics.
- `STUMINL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

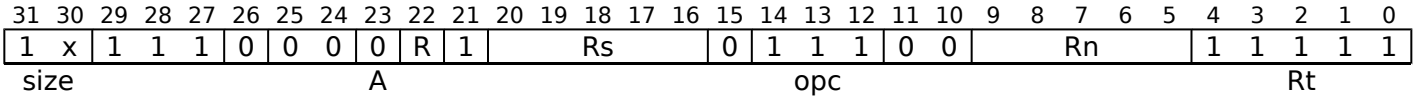
For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDUMIN, LDUMINA, LDUMINAL, LDUMINL. This means:

- The encodings in this description are named to match the encodings of LDUMIN, LDUMINA, LDUMINAL, LDUMINL.
- The description of LDUMIN, LDUMINA, LDUMINAL, LDUMINL gives the operational pseudocode for this instruction.

Integer

(FEAT_LSE)



32-bit LDUMIN alias (size == 10 && R == 0)

```
STUMIN <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMIN <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

32-bit LDUMINL alias (size == 10 && R == 1)

```
STUMINL <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMINL <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDUMIN alias (size == 11 && R == 0)

```
STUMIN <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDUMIN <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

64-bit LDUMINL alias (size == 11 && R == 1)

```
STUMINL <Xs>, [<Xn|SP>]
```

is equivalent to

```
LDUMINL <Xs>, XZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xs>	Is the 64-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMIN, LDUMINA, LDUMINAL, LDUMINL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STUMINB, STUMINLB

Atomic unsigned minimum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- `STUMINB` does not have release semantics.
- `STUMINLB` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

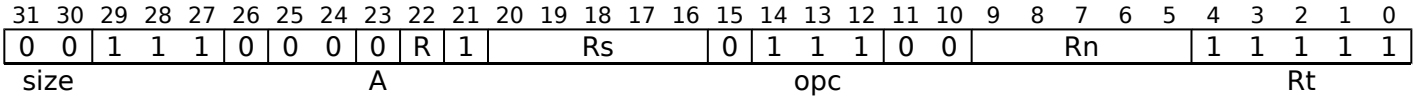
For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB. This means:

- The encodings in this description are named to match the encodings of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB.
- The description of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB gives the operational pseudocode for this instruction.

Integer

(FEAT_LSE)



No memory ordering (R == 0)

```
STUMINB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMINB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release (R == 1)

```
STUMINLB <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMINLB <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STUMINH, STUMINLH

Atomic unsigned minimum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- `STUMINH` does not have release semantics.
- `STUMINLH` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

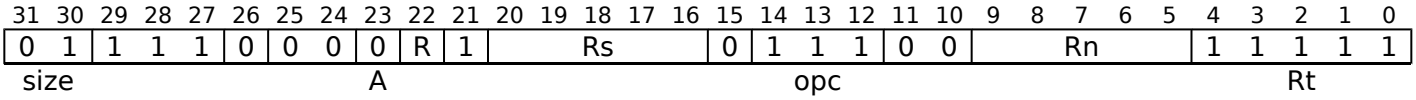
For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH. This means:

- The encodings in this description are named to match the encodings of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH.
- The description of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH gives the operational pseudocode for this instruction.

Integer

(FEAT_LSE)



No memory ordering (R == 0)

```
STUMINH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMINH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Release (R == 1)

```
STUMINLH <Ws>, [<Xn|SP>]
```

is equivalent to

```
LDUMINLH <Ws>, WZR, [<Xn|SP>]
```

and is always the preferred disassembly.

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register holding the data value to be operated on with the contents of the memory location, encoded in the "Rs" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

The description of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH gives the operational pseudocode for this instruction.

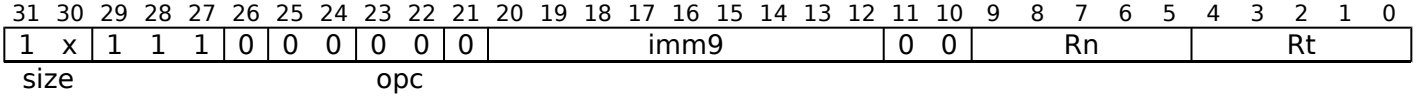
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STUR

Store Register (unscaled) calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about memory accesses, see *Load/Store addressing modes*.



32-bit (size == 10)

```
STUR <Wt>, [<Xn|SP>{, #<simm>}]
```

64-bit (size == 11)

```
STUR <Xt>, [<Xn|SP>{, #<simm>}]
```

```
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xt>	Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

integer datasize = 8 << scale;
boolean tag_checked = n != 31;
```

Operation

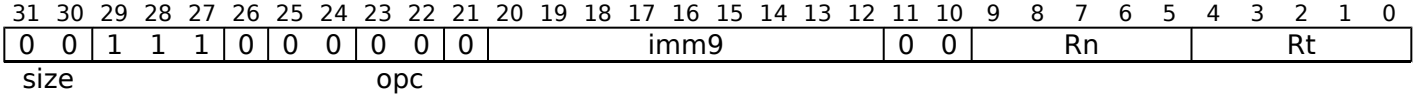
```
bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = X[t, datasize];
Mem[address, datasize DIV 8, AccType_NORMAL] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

STURB

Store Register Byte (unscaled) calculates an address from a base register value and an immediate offset, and stores a byte to the calculated address, from a 32-bit register. For information about memory accesses, see *Load/Store addressing modes*.



STURB <Wt>, [<Xn|SP>{, #<simm>}]

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Operation

```
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = X[t, 8];
Mem[address, 1, AccType_NORMAL] = data;
```

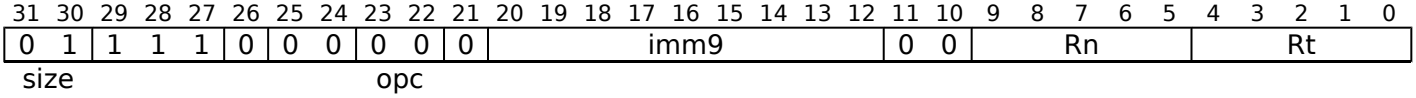
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STURH

Store Register Halfword (unscaled) calculates an address from a base register value and an immediate offset, and stores a halfword to the calculated address, from a 32-bit register. For information about memory accesses, see *Load/Store addressing modes*.



STURH <Wt>, [<Xn|SP>{, #<simm>}]

```
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Wt>	Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

boolean tag_checked = n != 31;
```

Operation

```
bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data = X[t, 16];
Mem[address, 2, AccType_NORMAL] = data;
```

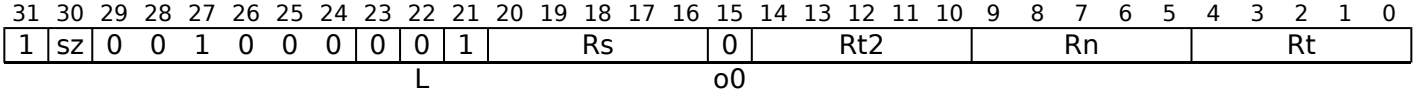
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STXP

Store Exclusive Pair of registers stores two 32-bit words or two 64-bit doublewords from two registers to a memory location if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. For information on single-copy atomicity and alignment requirements, see *Requirements for single-copy atomicity* and *Alignment of data accesses*. If a 64-bit pair Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. For information about memory accesses, see *Load/Store addressing modes*.



32-bit (sz == 0)

```
STXP <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{,#0}]
```

64-bit (sz == 1)

```
STXP <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);    // ignored by load/store single register
integer s = UInt(Rs);    // ignored by all loads and store-release

integer elsize = 32 << UInt(sz);
integer datasize = elsize * 2;
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t || (s == t2) then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STXP*.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0
If the operation updates memory.

1
If the operation fails to update memory.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(datasize) UNKNOWN;
else
    bits(datasize DIV 2) el1 = X[t, datasize DIV 2];
    bits(datasize DIV 2) el2 = X[t2, datasize DIV 2];
    data = if BigEndian(AccType_ATOMIC) then el1:el2 else el2:el1;
bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, dbytes) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, AccType_ATOMIC] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

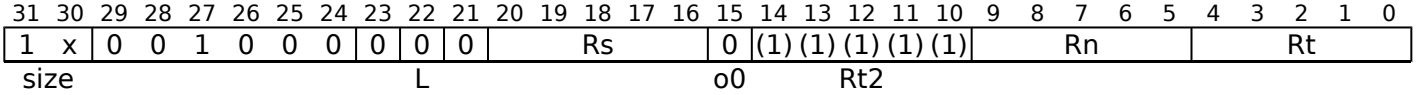
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STXR

Store Exclusive Register stores a 32-bit word or a 64-bit doubleword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. For information about memory accesses see *Load/Store addressing modes*.



32-bit (size == 10)

```
STXR <Ws>, <Wt>, [<Xn|SP>{,#0}]
```

64-bit (size == 11)

```
STXR <Ws>, <Xt>, [<Xn|SP>{,#0}]
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release

integer elsize = 8 << UInt(size);
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STXR*.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0
If the operation updates memory.

1
If the operation fails to update memory.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(elsize) data;
constant integer dbytes = elsize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(elsize) UNKNOWN;
else
    data = X[t, elsize];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, dbytes) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, dbytes, AccType_ATOMIC] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

Operational information

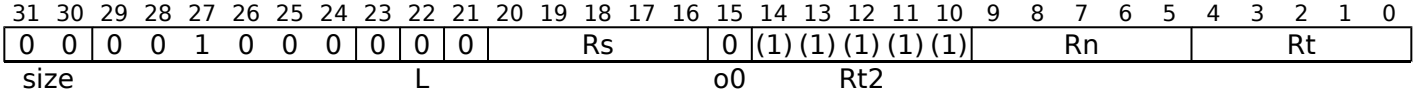
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STXRB

Store Exclusive Register Byte stores a byte from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. The memory access is atomic.

For information about memory accesses see *Load/Store addressing modes*.



STXRB <Ws>, <Wt>, [<Xn|SP>{,#0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release

boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STXRB*.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0
If the operation updates memory.

1
If the operation fails to update memory.

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(8) UNKNOWN;
else
    data = X[t, 8];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 1) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 1, AccType_ATOMIC] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

Operational information

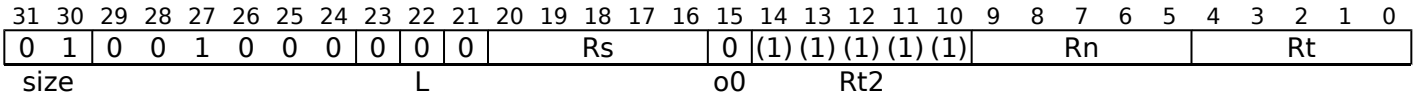
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STXRH

Store Exclusive Register Halfword stores a halfword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. The memory access is atomic.

For information about memory accesses see *Load/Store addressing modes*.



STXRH <Ws>, <Wt>, [<Xn|SP>{,#0}]

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);    // ignored by all loads and store-release

boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0
If the operation updates memory.

1
If the operation fails to update memory.

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(64) address;
bits(16) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
elsif rn_unknown then
    address = bits(64) UNKNOWN;
else
    address = X[n, 64];

if rt_unknown then
    data = bits(16) UNKNOWN;
else
    data = X[t, 16];

bit status = '1';
// Check whether the Exclusives monitors are set to include the
// physical memory locations corresponding to virtual address
// range [address, address+dbytes-1].
if AArch64.ExclusiveMonitorsPass(address, 2) then
    // This atomic write will be rejected if it does not refer
    // to the same physical locations after address translation.
    Mem[address, 2, AccType_ATOMIC] = data;
    status = ExclusiveMonitorsStatus();
X[s, 32] = ZeroExtend(status, 32);
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STZ2G

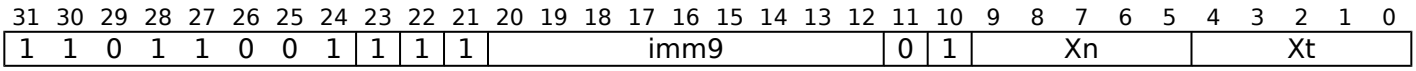
Store Allocation Tags, Zeroing stores an Allocation Tag to two Tag granules of memory, zeroing the associated data locations. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: Post-index , Pre-index and Signed offset

Post-index

(FEAT_MTE)

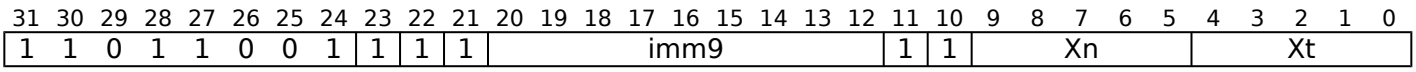


STZ2G <Xt|SP>, [<Xn|SP>], #<simm>

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;
```

Pre-index

(FEAT_MTE)

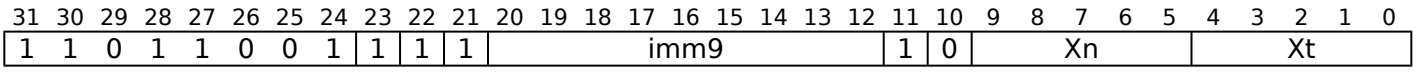


STZ2G <Xt|SP>, [<Xn|SP>, #<simm>]!

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;
```

Signed offset

(FEAT_MTE)



STZ2G <Xt|SP>, [<Xn|SP>{, #<simm>}]

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;
```

Assembler Symbols

<Xt|SP>	Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Xt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
<simm>	Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

Operation

```
bits(64) address;
bits(64) data = if t == 31 then SP[] else X[t, 64];
bits(4) tag = AArch64.AllocationTagFromAddress(data);

SetTagCheckedInstruction(FALSE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

if address != Align(address, TAG_GRANULE) then
    AArch64.Abort(address, AlignmentFault(AccType_NORMAL, TRUE, FALSE));

Mem[address, TAG_GRANULE, AccType_NORMAL] = Zeros(TAG_GRANULE * 8);
Mem[address+TAG_GRANULE, TAG_GRANULE, AccType_NORMAL] = Zeros(TAG_GRANULE * 8);

AArch64.MemTag[address, AccType_NORMAL] = tag;
AArch64.MemTag[address+TAG_GRANULE, AccType_NORMAL] = tag;

if writeback then
    if postindex then
        address = address + offset;

    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STZG

Store Allocation Tag, Zeroing stores an Allocation Tag to memory, zeroing the associated data location. The address used for the store is calculated from the base register and an immediate signed offset scaled by the Tag granule. The Allocation Tag is calculated from the Logical Address Tag in the source register.

This instruction generates an Unchecked access.

It has encodings from 3 classes: Post-index , Pre-index and Signed offset

Post-index

(FEAT_MTE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	0	1	1	imm9									0	1	Xn					Xt				

STZG <Xt|SP>, [<Xn|SP>], #<simm>

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = TRUE;
```

Pre-index

(FEAT_MTE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	0	1	1	imm9									1	1	Xn					Xt				

STZG <Xt|SP>, [<Xn|SP>, #<simm>]!

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = TRUE;
boolean postindex = FALSE;
```

Signed offset

(FEAT_MTE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	0	0	1	0	1	1	imm9									1	0	Xn					Xt				

STZG <Xt|SP>, [<Xn|SP>{, #<simm>}]

```
if !HaveMTEExt() then UNDEFINED;
integer n = UInt(Xn);
integer t = UInt(Xt);
bits(64) offset = LSL(SignExtend(imm9, 64), LOG2_TAG_GRANULE);
boolean writeback = FALSE;
boolean postindex = FALSE;
```

Assembler Symbols

<Xt|SP>	Is the 64-bit name of the general-purpose source register or stack pointer, encoded in the "Xt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.
<simm>	Is the optional signed immediate offset, a multiple of 16 in the range -4096 to 4080, defaulting to 0 and encoded in the "imm9" field.

Operation

```
bits(64) address;

SetTagCheckedInstruction(FALSE);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

if address != Align(address, TAG_GRANULE) then
    AArch64.Abort(address, AlignmentFault(AccType_NORMAL, TRUE, FALSE));

Mem[address, TAG_GRANULE, AccType_NORMAL] = Zeros(TAG_GRANULE * 8);

bits(64) data = if t == 31 then SP[] else X[t, 64];
bits(4) tag = AArch64.AllocationTagFromAddress(data);
AArch64.MemTag[address, AccType_NORMAL] = tag;

if writeback then
    if postindex then
        address = address + offset;

    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

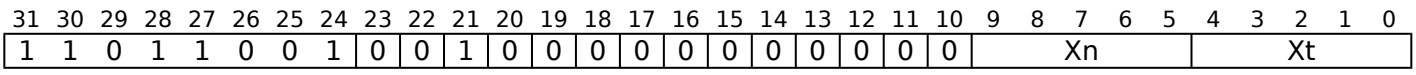
STZGM

Store Tag and Zero Multiple writes a naturally aligned block of N Allocation Tags and stores zero to the associated data locations, where the size of N is identified in DCZID_EL0.BS, and the Allocation Tag is taken from the source register bits<3:0>.

This instruction is UNDEFINED at EL0.

This instruction generates an Unchecked access.

Integer
(FEAT_MTE2)



STZGM <Xt>, [<Xn|SP>]

```
if !HaveMTE2Ext() then UNDEFINED;
integer t = UInt(Xt);
integer n = UInt(Xn);
```

Assembler Symbols

<Xt>	Is the 64-bit name of the general-purpose source register, encoded in the "Xt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Xn" field.

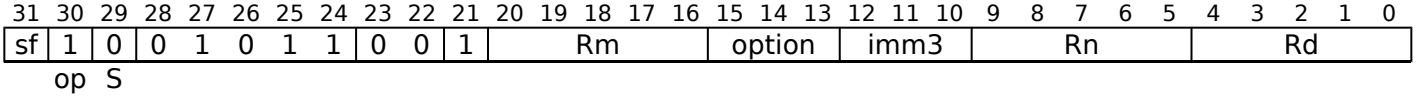
Operation

```
if PSTATE.EL == EL0 then
    UNDEFINED;

bits(64) data = X[t, 64];
bits(4) tag = data<3:0>;
bits(64) address;
if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

integer size = 4 * (2 ^ (UInt(DCZID_EL0.BS)));
address = Align(address, size);
integer count = size >> LOG2_TAG_GRANULE;

for i = 0 to count-1
    AArch64.MemTag[address, AccType_NORMAL] = tag;
    Mem[address, TAG_GRANULE, AccType_NORMAL] = Zeros(8 * TAG_GRANULE);
    address = address + TAG_GRANULE;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SUB (extended register)

Subtract (extended register) subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword.



32-bit (sf == 0)

```
SUB <Wd|WSP>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

64-bit (sf == 1)

```
SUB <Xd|SP>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then UNDEFINED;
```

Assembler Symbols

<Wd|WSP> Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Wn|WSP> Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd|SP> Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in “option”:

option	<R>
00x	W
010	W
x11	X
10x	W
110	W

<m> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in “option”:

option	<extend>
000	UXTB
001	UXTH
010	LSL|UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If "Rd" or "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in “option”:

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL|UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If "Rd" or "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[]<datasize-1:0> else X[n, datasize];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift, datasize);

operand2 = NOT(operand2);
(result, -) = AddWithCarry(operand1, operand2, '1');

if d == 31 then
    SP[] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

Operational information

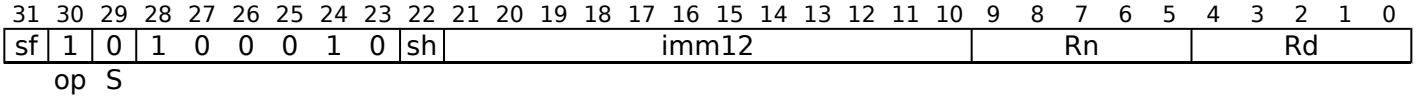
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SUB (immediate)

Subtract (immediate) subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register.



32-bit (sf == 0)

```
SUB <Wd|WSP>, <Wn|WSP>, #<imm>{, <shift>}
```

64-bit (sf == 1)

```
SUB <Xd|SP>, <Xn|SP>, #<imm>{, <shift>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;

case sh of
    when '0' imm = ZeroExtend(imm12, datasize);
    when '1' imm = ZeroExtend(imm12:Zeros(12), datasize);
```

Assembler Symbols

<Wd|WSP>	Is the 32-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Wn|WSP>	Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<Xd|SP>	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xn|SP>	Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<imm>	Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
<shift>	Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #12

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[]<datasize-1:0> else X[n, datasize];
bits(datasize) operand2;

operand2 = NOT(imm);
(result, -) = AddWithCarry(operand1, operand2, '1');

if d == 31 then
    SP[] = ZeroExtend(result, 64);
else
    X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

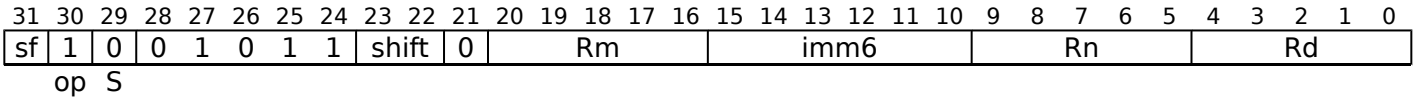
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SUB (shifted register)

Subtract (shifted register) subtracts an optionally-shifted register value from a register value, and writes the result to the destination register.

This instruction is used by the alias NEG (shifted register).



32-bit (sf == 0)

```
SUB <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit (sf == 1)

```
SUB <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;

if shift == '11' then UNDEFINED;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Alias Conditions

Alias	Is preferred when
NEG (shifted register)	`Rn == '11111'`

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);

operand2 = NOT(operand2);
(result, -) = AddWithCarry(operand1, operand2, '1');

X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

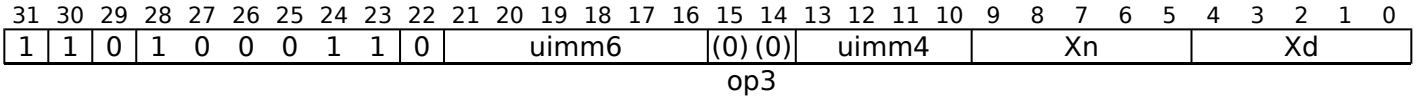
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SUBG

Subtract with Tag subtracts an immediate value scaled by the Tag granule from the address in the source register, modifies the Logical Address Tag of the address using an immediate value, and writes the result to the destination register. Tags specified in GCR_EL1.Exclude are excluded from the possible outputs when modifying the Logical Address Tag.

Integer
(FEAT_MTE)



SUBG <Xd|SP>, <Xn|SP>, #<uimm6>, #<uimm4>

```
if !HaveMTEExt() then UNDEFINED;
integer d = UInt(Xd);
integer n = UInt(Xn);
bits(64) offset = LSL(ZeroExtend(uimm6, 64), LOG2_TAG_GRANULE);
```

Assembler Symbols

<Xd|SP>	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Xd" field.
<Xn|SP>	Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Xn" field.
<uimm6>	Is an unsigned immediate, a multiple of 16 in the range 0 to 1008, encoded in the "uimm6" field.
<uimm4>	Is an unsigned immediate, in the range 0 to 15, encoded in the "uimm4" field.

Operation

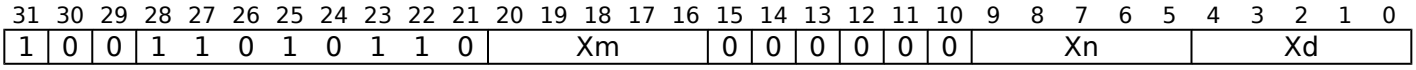
```
bits(64) operand1 = if n == 31 then SP[] else X[n, 64];
bits(4) start_tag = AArch64.AllocationTagFromAddress(operand1);
bits(16) exclude = GCR_EL1.Exclude;
bits(64) result;
bits(4) rtag;

if AArch64.AllocationTagAccessIsEnabled(AccType_NORMAL) then
    rtag = AArch64.ChooseNonExcludedTag(start_tag, uimm4, exclude);
else
    rtag = '0000';

(result, -) = AddWithCarry(operand1, NOT(offset), '1');

result = AArch64.AddressWithAllocationTag(result, AccType_NORMAL, rtag);

if d == 31 then
    SP[] = result;
else
    X[d, 64] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SUBP

Subtract Pointer subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, sign-extends the result to 64-bits, and writes the result to the destination register.

Integer
(FEAT_MTE)



SUBP <Xd>, <Xn|SP>, <Xm|SP>

```
if !HaveMTEExt() then UNDEFINED;
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);
```

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Xd" field.
<Xn|SP>	Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.
<Xm|SP>	Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Xm" field.

Operation

```
bits(64) operand1 = if n == 31 then SP[] else X[n, 64];
bits(64) operand2 = if m == 31 then SP[] else X[m, 64];
operand1 = SignExtend(operand1<55:0>, 64);
operand2 = SignExtend(operand2<55:0>, 64);

bits(64) result;

operand2 = NOT(operand2);
(result, -) = AddWithCarry(operand1, operand2, '1');

X[d, 64] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

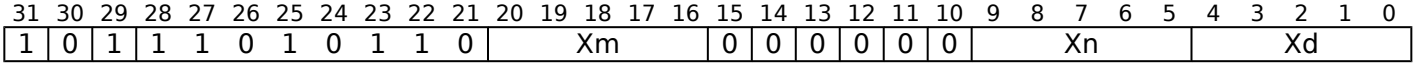
SUBPS

Subtract Pointer, setting Flags subtracts the 56-bit address held in the second source register from the 56-bit address held in the first source register, sign-extends the result to 64-bits, and writes the result to the destination register. It updates the condition flags based on the result of the subtraction.

This instruction is used by the alias CMPP.

Integer

(FEAT_MTE)



SUBPS <Xd>, <Xn|SP>, <Xm|SP>

```
if !HaveMTEExt() then UNDEFINED;
integer d = UInt(Xd);
integer n = UInt(Xn);
integer m = UInt(Xm);
```

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Xd" field.
<Xn|SP>	Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Xn" field.
<Xm|SP>	Is the 64-bit name of the second general-purpose source register or stack pointer, encoded in the "Xm" field.

Alias Conditions

Alias	Is preferred when
CMPP	`S == '1' && Xd == '11111'`

Operation

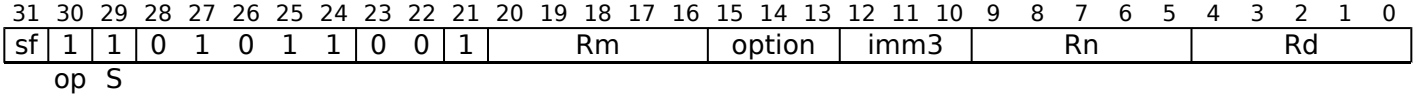
```
bits(64) operand1 = if n == 31 then SP[] else X[n, 64];
bits(64) operand2 = if m == 31 then SP[] else X[m, 64];
operand1 = SignExtend(operand1<55:0>, 64);
operand2 = SignExtend(operand2<55:0>, 64);

bits(64) result;
bits(4) nzcv;

operand2 = NOT(operand2);
(result, nzcv) = AddWithCarry(operand1, operand2, '1');

PSTATE.<N,Z,C,V> = nzcv;
X[d, 64] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SUBS (extended register)

Subtract (extended register), setting flags, subtracts a sign or zero-extended register value, followed by an optional left shift amount, from a register value, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword. It updates the condition flags based on the result.

This instruction is used by the alias CMP (extended register).



32-bit (sf == 0)

```
SUBS <Wd>, <Wn|WSP>, <Wm>{, <extend> {#<amount>}}
```

64-bit (sf == 1)

```
SUBS <Xd>, <Xn|SP>, <R><m>{, <extend> {#<amount>}}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then UNDEFINED;
```

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn|WSP> Is the 32-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn|SP> Is the 64-bit name of the first source general-purpose register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in “option”:

option	<R>
00x	W
010	W
x11	X
10x	W
110	W

<m> Is the number [0-30] of the second general-purpose source register or the name ZR (31), encoded in the "Rm" field.

<extend> For the 32-bit variant: is the extension to be applied to the second source operand, encoded in “option”:

option	<extend>
000	UXTB
001	UXTH
010	LSL|UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If "Rn" is '11111' (WSP) and "option" is '010' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTW when "option" is '010'.

For the 64-bit variant: is the extension to be applied to the second source operand, encoded in “option”:

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	LSL|UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

If "Rn" is '11111' (SP) and "option" is '011' then LSL is preferred, but may be omitted when "imm3" is '000'. In all other cases <extend> is required and must be UXTX when "option" is '011'.

<amount> Is the left shift amount to be applied after extension in the range 0 to 4, defaulting to 0, encoded in the "imm3" field. It must be absent when <extend> is absent, is required when <extend> is LSL, and is optional when <extend> is present but not LSL.

Alias Conditions

Alias	Is preferred when
CMP (extended register)	`Rd == '11111'`

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[]<datasize-1:0> else X[n, datasize];
bits(datasize) operand2 = ExtendReg(m, extend_type, shift, datasize);
bits(4) nzcv;

operand2 = NOT(operand2);
(result, nzcv) = AddWithCarry(operand1, operand2, '1');

PSTATE.<N,Z,C,V> = nzcv;

X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

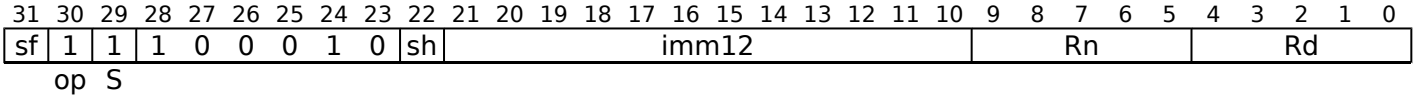
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SUBS (immediate)

Subtract (immediate), setting flags, subtracts an optionally-shifted immediate value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the alias CMP (immediate).



32-bit (sf == 0)

```
SUBS <Wd>, <Wn|WSP>, #<imm>{, <shift>}
```

64-bit (sf == 1)

```
SUBS <Xd>, <Xn|SP>, #<imm>{, <shift>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;
bits(datasize) imm;

case sh of
    when '0' imm = ZeroExtend(imm12, datasize);
    when '1' imm = ZeroExtend(imm12:Zeros(12), datasize);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn|WSP>	Is the 32-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn|SP>	Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<imm>	Is an unsigned immediate, in the range 0 to 4095, encoded in the "imm12" field.
<shift>	Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #12

Alias Conditions

Alias	Is preferred when
CMP (immediate)	`Rd == '11111'`

Operation

```
bits(datasize) result;
bits(datasize) operand1 = if n == 31 then SP[]<datasize-1:0> else X[n, datasize];
bits(datasize) operand2;
bits(4) nzcv;

operand2 = NOT(imm);
(result, nzcv) = AddWithCarry(operand1, operand2, '1');

PSTATE.<N,Z,C,V> = nzcv;

X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

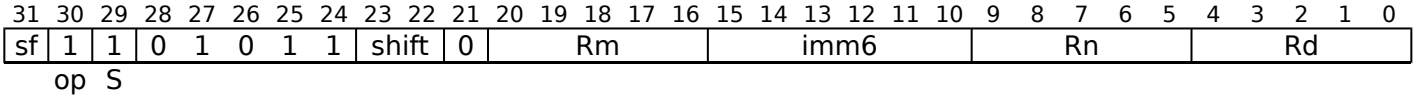
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SUBS (shifted register)

Subtract (shifted register), setting flags, subtracts an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

This instruction is used by the aliases CMP (shifted register), and NEGS.



32-bit (sf == 0)

```
SUBS <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit (sf == 1)

```
SUBS <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;

if shift == '11' then UNDEFINED;
if sf == '0' && imm6<5> == '1' then UNDEFINED;

ShiftType shift_type = DecodeShift(shift);
integer shift_amount = UInt(imm6);
```

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift type to be applied to the second source operand, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR
11	RESERVED

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field.

Alias Conditions

Alias	Is preferred when
CMP (shifted register)	`Rd == '11111'`
NEGS	`Rn == '11111' && Rd != '11111'`

Operation

```
bits(datasize) result;
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = ShiftReg(m, shift_type, shift_amount, datasize);
bits(4) nzcv;

operand2 = NOT(operand2);
(result, nzcv) = AddWithCarry(operand1, operand2, '1');

PSTATE.<N,Z,C,V> = nzcv;

X[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

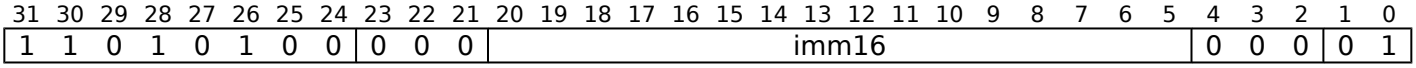
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SVC

Supervisor Call causes an exception to be taken to EL1.

On executing an SVC instruction, the PE records the exception as a Supervisor Call exception in *ESR_ELx*, using the EC value 0x15, and the value of the immediate argument.



SVC #<imm>

```
// Empty.
```

Assembler Symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation

```
AArch64.CheckForSVCTrap(imm16);
AArch64.CallSupervisor(imm16);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SWP, SWPA, SWPAL, SWPL

Swap word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

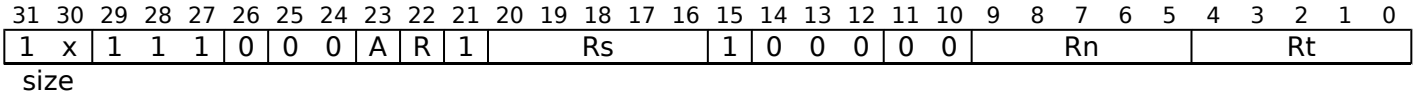
- If the destination register is not one of WZR or XZR, `SWPA` and `SWPAL` load from memory with acquire semantics.
- `SWPL` and `SWPAL` store to memory with release semantics.
- `SWP` has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(FEAT_LSE)



32-bit SWP (size == 10 && A == 0 && R == 0)

```
SWP <Ws>, <Wt>, [<Xn|SP>]
```

32-bit SWPA (size == 10 && A == 1 && R == 0)

```
SWPA <Ws>, <Wt>, [<Xn|SP>]
```

32-bit SWPAL (size == 10 && A == 1 && R == 1)

```
SWPAL <Ws>, <Wt>, [<Xn|SP>]
```

32-bit SWPL (size == 10 && A == 0 && R == 1)

```
SWPL <Ws>, <Wt>, [<Xn|SP>]
```

64-bit SWP (size == 11 && A == 0 && R == 0)

```
SWP <Xs>, <Xt>, [<Xn|SP>]
```

64-bit SWPA (size == 11 && A == 1 && R == 0)

```
SWPA <Xs>, <Xt>, [<Xn|SP>]
```

64-bit SWPAL (size == 11 && A == 1 && R == 1)

```
SWPAL <Xs>, <Xt>, [<Xn|SP>]
```

64-bit SWPL (size == 11 && A == 0 && R == 1)

```
SWPL <Xs>, <Xt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xs>	Is the 64-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
<Xt>	Is the 64-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) data;
bits(datasize) store_value;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

store_value = X[s, datasize];
data = MemAtomic(address, MemAtomicOp_SWP, store_value, ldacctype, stacctype);
X[t, regsize] = ZeroExtend(data, regsize);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SWPB, SWPAB, SWPALB, SWPLB

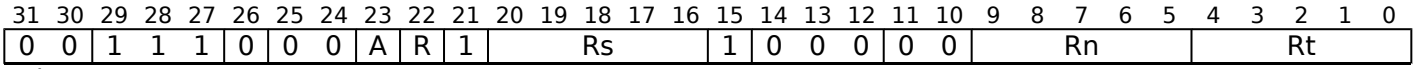
Swap byte in memory atomically loads an 8-bit byte from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, SWPAB and SWPALB load from memory with acquire semantics.
- SWPLB and SWPALB store to memory with release semantics.
- SWPB has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer
(FEAT_LSE)



SWPAB (A == 1 && R == 0)

```
SWPAB <Ws>, <Wt>, [<Xn|SP>]
```

SWPALB (A == 1 && R == 1)

```
SWPALB <Ws>, <Wt>, [<Xn|SP>]
```

SWPB (A == 0 && R == 0)

```
SWPB <Ws>, <Wt>, [<Xn|SP>]
```

SWPLB (A == 0 && R == 1)

```
SWPLB <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(8) data;
bits(8) store_value;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

store_value = X[s, 8];
data = MemAtomic(address, MemAtomicOp_SWP, store_value, ldacctype, stacctype);
X[t, 32] = ZeroExtend(data, 32);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SWPH, SWPAH, SWPALH, SWPLH

Swap halfword in memory atomically loads a 16-bit halfword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

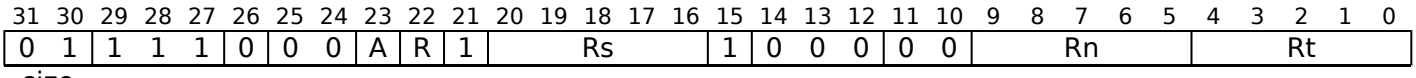
- If the destination register is not WZR, SWPAH and SWPALH load from memory with acquire semantics.
- SWPLH and SWPALH store to memory with release semantics.
- SWPH has neither acquire nor release semantics.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

Integer

(FEAT_LSE)



SWPAH (A == 1 && R == 0)

```
SWPAH <Ws>, <Wt>, [<Xn|SP>]
```

SWPALH (A == 1 && R == 1)

```
SWPALH <Ws>, <Wt>, [<Xn|SP>]
```

SWPH (A == 0 && R == 0)

```
SWPH <Ws>, <Wt>, [<Xn|SP>]
```

SWPLH (A == 0 && R == 1)

```
SWPLH <Ws>, <Wt>, [<Xn|SP>]
```

```
if !HaveAtomicExt() then UNDEFINED;

integer t = UInt(Rt);
integer n = UInt(Rn);
integer s = UInt(Rs);

AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
boolean tag_checked = n != 31;
```

Assembler Symbols

<Ws>	Is the 32-bit name of the general-purpose register to be stored, encoded in the "Rs" field.
<Wt>	Is the 32-bit name of the general-purpose register to be loaded, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(16) data;
bits(16) store_value;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

store_value = X[s, 16];
data = MemAtomic(address, MemAtomicOp_SWP, store_value, ldacctype, stacctype);
X[t, 32] = ZeroExtend(data, 32);
```

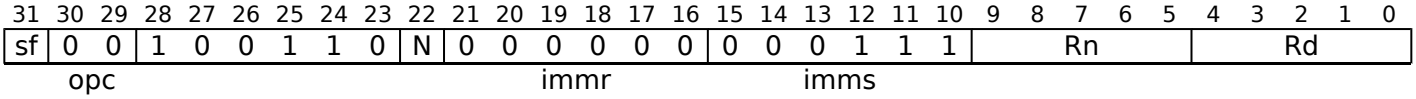
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SXTB

Signed Extend Byte extracts an 8-bit value from a register, sign-extends it to the size of the register, and writes the result to the destination register.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.



32-bit (sf == 0 && N == 0)

```
SXTB <Wd>, <Wn>
```

is equivalent to

```
SBFM <Wd>, <Wn>, #0, #7
```

and is always the preferred disassembly.

64-bit (sf == 1 && N == 1)

```
SXTB <Xd>, <Wn>
```

is equivalent to

```
SBFM <Xd>, <Xn>, #0, #7
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

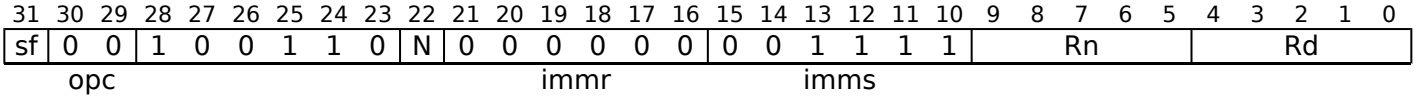
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SXTH

Sign Extend Halfword extracts a 16-bit value, sign-extends it to the size of the register, and writes the result to the destination register.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.



32-bit (sf == 0 && N == 0)

```
SXTH <Wd>, <Wn>
```

is equivalent to

```
SBFM <Wd>, <Wn>, #0, #15
```

and is always the preferred disassembly.

64-bit (sf == 1 && N == 1)

```
SXTH <Xd>, <Wn>
```

is equivalent to

```
SBFM <Xd>, <Xn>, #0, #15
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

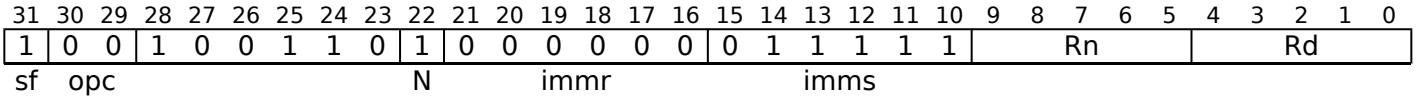
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SXTW

Sign Extend Word sign-extends a word to the size of the register, and writes the result to the destination register.

This is an alias of SBFM. This means:

- The encodings in this description are named to match the encodings of SBFM.
- The description of SBFM gives the operational pseudocode for this instruction.



64-bit

```
SXTW <Xd>, <Wn>
```

is equivalent to

```
SBFM <Xd>, <Xn>, #0, #31
```

and is always the preferred disassembly.

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of SBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

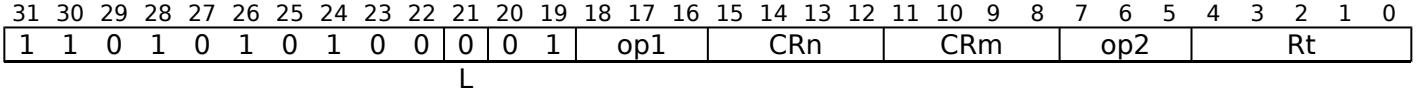
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SYS

System instruction. For more information, see *Op0 equals 0b01, cache maintenance, TLB maintenance, and address translation instructions* for the encodings of System instructions.

This instruction is used by the aliases AT, BRB, CFP, CPP, DC, DVP, IC, and TLBI.



SYS #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}

```
AArch64.CheckSystemAccess('01', op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);

integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
```

Assembler Symbols

<op1>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
<Cn>	Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
<Cm>	Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
<op2>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.
<Xt>	Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

Alias Conditions

Alias	Is preferred when
AT	`CRn == '0111' && CRm == '100x' && SysOp(op1,'0111',CRm,op2) == Sys_AT`
BRB	`op1 == '001' && CRn == '0111' && CRm == '0010' && SysOp('001','0111','0010',op2) == Sys_BRB`
CFP	`op1 == '011' && CRn == '0111' && CRm == '0011' && op2 == '100'`
CPP	`op1 == '011' && CRn == '0111' && CRm == '0011' && op2 == '111'`
DC	`CRn == '0111' && SysOp(op1,'0111',CRm,op2) == Sys_DC`
DVP	`op1 == '011' && CRn == '0111' && CRm == '0011' && op2 == '101'`
IC	`CRn == '0111' && SysOp(op1,'0111',CRm,op2) == Sys_IC`
TLBI	`CRn == '100x' && SysOp(op1,CRn,CRm,op2) == Sys_TLBI`

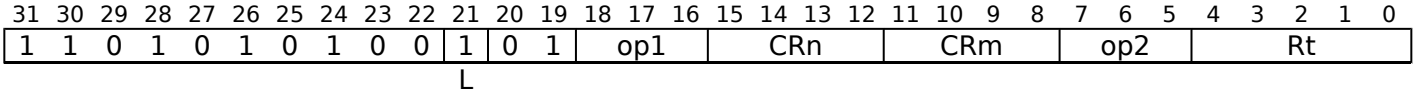
Operation

```
AArch64.SysInstr(1, sys_op1, sys_crn, sys_crm, sys_op2, t);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SYSL

System instruction with result. For more information, see *Op0 equals 0b01, cache maintenance, TLB maintenance, and address translation instructions* for the encodings of System instructions.



SYSL <Xt>, #<op1>, <Cn>, <Cm>, #<op2>

```
AArch64.CheckSystemAccess('01', op1, CRn, CRm, op2, Rt, L);

integer t = UInt(Rt);

integer sys_op1 = UInt(op1);
integer sys_op2 = UInt(op2);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
```

Assembler Symbols

<Xt>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.
<op1>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
<Cn>	Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
<Cm>	Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
<op2>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

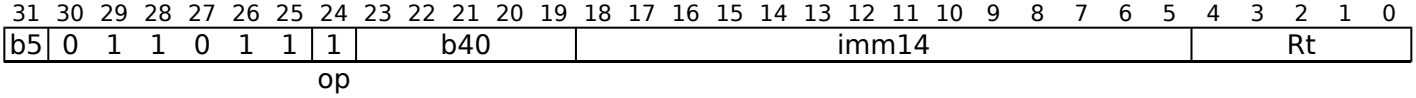
Operation

```
// No architecturally defined instructions here.
AArch64.SysInstrWithResult(1, sys_op1, sys_crn, sys_crm, sys_op2, t);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

TBNZ

Test bit and Branch if Nonzero compares the value of a bit in a general-purpose register with zero, and conditionally branches to a label at a PC-relative offset if the comparison is not equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.



TBNZ <R><t>, #<imm>, <label>

```
integer t = UInt(Rt);

integer datasize = if b5 == '1' then 64 else 32;
integer bit_pos = UInt(b5:b40);
bits(64) offset = SignExtend(imm14:'00', 64);
```

Assembler Symbols

<R> Is a width specifier, encoded in “b5”:

b5	<R>
0	W
1	X

In assembler source code an 'X' specifier is always permitted, but a 'W' specifier is only permitted when the bit number is less than 32.

<t> Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the "Rt" field.

<imm> Is the bit number to be tested, in the range 0 to 63, encoded in "b5:b40".

<label> Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as "imm14" times 4.

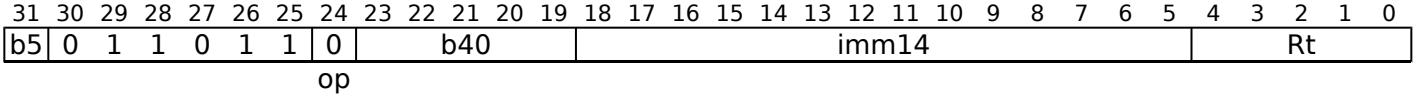
Operation

```
bits(datasize) operand = X[t, datasize];
if operand<bit_pos> == op then
    BranchTo(PC[] + offset, BranchType_DIR, TRUE);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

TBZ

Test bit and Branch if Zero compares the value of a test bit with zero, and conditionally branches to a label at a PC-relative offset if the comparison is equal. It provides a hint that this is not a subroutine call or return. This instruction does not affect condition flags.



TBZ <R><t>, #<imm>, <label>

```
integer t = UInt(Rt);

integer datasize = if b5 == '1' then 64 else 32;
integer bit_pos = UInt(b5:b40);
bits(64) offset = SignExtend(imm14:'00', 64);
```

Assembler Symbols

<R> Is a width specifier, encoded in “b5”:

b5	<R>
0	W
1	X

In assembler source code an 'X' specifier is always permitted, but a 'W' specifier is only permitted when the bit number is less than 32.

<t> Is the number [0-30] of the general-purpose register to be tested or the name ZR (31), encoded in the "Rt" field.

<imm> Is the bit number to be tested, in the range 0 to 63, encoded in "b5:b40".

<label> Is the program label to be conditionally branched to. Its offset from the address of this instruction, in the range +/-32KB, is encoded as "imm14" times 4.

Operation

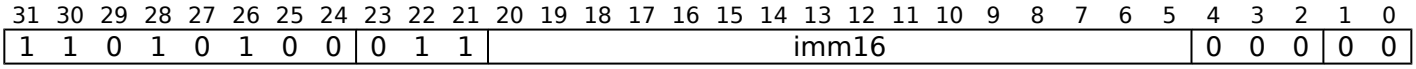
```
bits(datasize) operand = X[t, datasize];
if operand<bit_pos> == op then
    BranchTo(PC[] + offset, BranchType_DIR, TRUE);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

TCANCEL

This instruction exits Transactional state and discards all state modifications that were performed transactionally. Execution continues at the instruction that follows the TSTART instruction of the outer transaction. The destination register of the TSTART instruction of the outer transaction is written with the immediate operand of TCANCEL.

System
(FEAT_TME)



TCANCEL #<imm>

```
if !HaveTME() then UNDEFINED;
boolean retry = (imm16<15> == '1');
bits(15) reason = imm16<14:0>;
```

Assembler Symbols

<imm> Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field.

Operation

```
CheckTMEEnabled();

if TSTATE.depth > 0 then
    FailTransaction(TMFailure_CNCL, retry, FALSE, reason);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

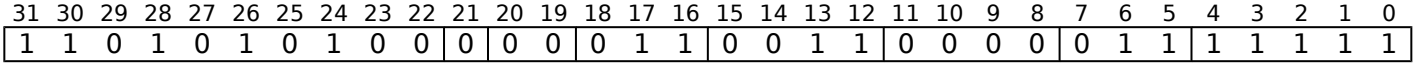
TCOMMIT

This instruction commits the current transaction. If the current transaction is an outer transaction, then Transactional state is exited, and all state modifications performed transactionally are committed to the architectural state. TCOMMIT takes no inputs and returns no value.

Execution of TCOMMIT is UNDEFINED in Non-transactional state.

System

(FEAT_TME)



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	0	0	0	0	0	1	1	1	1	1	1	1

TCOMMIT

```
if !HaveTME() then UNDEFINED;
```

Operation

```
CheckTMEEnabled();

if TSTATE.depth == 0 then
    UNDEFINED;

if TSTATE.depth == 1 then
    CommitTransactionalWrites();
    ClearExclusiveLocal(ProcessorID());

TSTATE.depth = TSTATE.depth - 1;
```

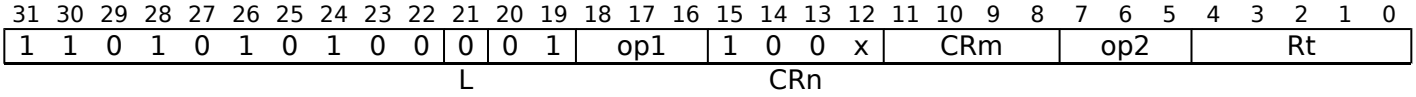
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

TLBI

TLB Invalidate operation. For more information, see *op0==0b01, cache maintenance, TLB maintenance, and address translation instructions*.

This is an alias of SYS. This means:

- The encodings in this description are named to match the encodings of SYS.
- The description of SYS gives the operational pseudocode for this instruction.



TLBI <tlbi_op>{, <Xt>}

is equivalent to

SYS #<op1>, <Cn>, <Cm>, #<op2>{, <Xt>}

and is the preferred disassembly when `SysOp(op1,CRn,CRm,op2) == Sys_TLBI`.

Assembler Symbols

<op1>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.
<Cn>	Is a name 'Cn', with 'n' in the range 0 to 15, encoded in the "CRn" field.
<Cm>	Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.
<op2>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.
<tlbi_op>	Is a TLBI instruction name, as listed for the TLBI system instruction group, encoded in “op1:CRn:CRm:op2”:

op1	CRn	CRm	op2	<tlbi_op>	Architectural Feature
000	1000	0001	000	VMALLE1OS	FEAT_TLBIOS
000	1000	0001	001	VAE1OS	FEAT_TLBIOS
000	1000	0001	010	ASIDE1OS	FEAT_TLBIOS
000	1000	0001	011	VAAE1OS	FEAT_TLBIOS
000	1000	0001	101	VALE1OS	FEAT_TLBIOS
000	1000	0001	111	VAALE1OS	FEAT_TLBIOS
000	1000	0010	001	RVAE1IS	FEAT_TLBIRANGE
000	1000	0010	011	RVAAE1IS	FEAT_TLBIRANGE
000	1000	0010	101	RVALE1IS	FEAT_TLBIRANGE
000	1000	0010	111	RVAALE1IS	FEAT_TLBIRANGE
000	1000	0011	000	VMALLE1IS	-
000	1000	0011	001	VAE1IS	-
000	1000	0011	010	ASIDE1IS	-
000	1000	0011	011	VAAE1IS	-
000	1000	0011	101	VALE1IS	-
000	1000	0011	111	VAALE1IS	-
000	1000	0101	001	RVAE1OS	FEAT_TLBIRANGE
000	1000	0101	011	RVAAE1OS	FEAT_TLBIRANGE
000	1000	0101	101	RVALE1OS	FEAT_TLBIRANGE
000	1000	0101	111	RVAALE1OS	FEAT_TLBIRANGE
000	1000	0110	001	RVAE1	FEAT_TLBIRANGE
000	1000	0110	011	RVAAE1	FEAT_TLBIRANGE
000	1000	0110	101	RVALE1	FEAT_TLBIRANGE
000	1000	0110	111	RVAALE1	FEAT_TLBIRANGE
000	1000	0111	000	VMALLE1	-
000	1000	0111	001	VAE1	-
000	1000	0111	010	ASIDE1	-
000	1000	0111	011	VAAE1	-
000	1000	0111	101	VALE1	-
000	1000	0111	111	VAALE1	-
000	1001	0001	000	VMALLE1OSNXS	FEAT_XS
000	1001	0001	001	VAE1OSNXS	FEAT_XS
000	1001	0001	010	ASIDE1OSNXS	FEAT_XS
000	1001	0001	011	VAAE1OSNXS	FEAT_XS
000	1001	0001	101	VALE1OSNXS	FEAT_XS
000	1001	0001	111	VAALE1OSNXS	FEAT_XS
000	1001	0010	001	RVAE1ISNXS	FEAT_XS
000	1001	0010	011	RVAAE1ISNXS	FEAT_XS
000	1001	0010	101	RVALE1ISNXS	FEAT_XS
000	1001	0010	111	RVAALE1ISNXS	FEAT_XS
000	1001	0011	000	VMALLE1ISNXS	FEAT_XS
000	1001	0011	001	VAE1ISNXS	FEAT_XS
000	1001	0011	010	ASIDE1ISNXS	FEAT_XS
000	1001	0011	011	VAAE1ISNXS	FEAT_XS
000	1001	0011	101	VALE1ISNXS	FEAT_XS
000	1001	0011	111	VAALE1ISNXS	FEAT_XS
000	1001	0101	001	RVAE1OSNXS	FEAT_XS
000	1001	0101	011	RVAAE1OSNXS	FEAT_XS
000	1001	0101	101	RVALE1OSNXS	FEAT_XS
000	1001	0101	111	RVAALE1OSNXS	FEAT_XS
000	1001	0110	001	RVAE1NXS	FEAT_XS
000	1001	0110	011	RVAAE1NXS	FEAT_XS
000	1001	0110	101	RVALE1NXS	FEAT_XS
000	1001	0110	111	RVAALE1NXS	FEAT_XS
000	1001	0111	000	VMALLE1NXS	FEAT_XS
000	1001	0111	001	VAE1NXS	FEAT_XS
000	1001	0111	010	ASIDE1NXS	FEAT_XS
000	1001	0111	011	VAAE1NXS	FEAT_XS
000	1001	0111	101	VALE1NXS	FEAT_XS
000	1001	0111	111	VAALE1NXS	FEAT_XS
100	1000	0000	001	IPAS2E1IS	-
100	1000	0000	010	RIPAS2E1IS	FEAT_TLBIRANGE
100	1000	0000	101	IPAS2LE1IS	-
100	1000	0000	110	RIPAS2LE1IS	FEAT_TLBIRANGE
100	1000	0001	000	ALLE2OS	FEAT_TLBIOS
100	1000	0001	001	VAE2OS	FEAT_TLBIOS
100	1000	0001	100	ALLE1OS	FEAT_TLBIOS
100	1000	0001	101	VALE2OS	FEAT_TLBIOS

op1	CRn	CRm	op2	<tlbi_op>	Architectural Feature
100	1000	0001	110	VMALLS12E1OS	FEAT_TLBIOS
100	1000	0010	001	RVAE2IS	FEAT_TLBIRANGE
100	1000	0010	101	RVALE2IS	FEAT_TLBIRANGE
100	1000	0011	000	ALLE2IS	-
100	1000	0011	001	VAE2IS	-
100	1000	0011	100	ALLE1IS	-
100	1000	0011	101	VALE2IS	-
100	1000	0011	110	VMALLS12E1IS	-
100	1000	0100	000	IPAS2E1OS	FEAT_TLBIOS
100	1000	0100	001	IPAS2E1	-
100	1000	0100	010	RIPAS2E1	FEAT_TLBIRANGE
100	1000	0100	011	RIPAS2E1OS	FEAT_TLBIRANGE
100	1000	0100	100	IPAS2LE1OS	FEAT_TLBIOS
100	1000	0100	101	IPAS2LE1	-
100	1000	0100	110	RIPAS2LE1	FEAT_TLBIRANGE
100	1000	0100	111	RIPAS2LE1OS	FEAT_TLBIRANGE
100	1000	0101	001	RVAE2OS	FEAT_TLBIRANGE
100	1000	0101	101	RVALE2OS	FEAT_TLBIRANGE
100	1000	0110	001	RVAE2	FEAT_TLBIRANGE
100	1000	0110	101	RVALE2	FEAT_TLBIRANGE
100	1000	0111	000	ALLE2	-
100	1000	0111	001	VAE2	-
100	1000	0111	100	ALLE1	-
100	1000	0111	101	VALE2	-
100	1000	0111	110	VMALLS12E1	-
100	1001	0000	001	IPAS2E1ISNXS	FEAT_XS
100	1001	0000	010	RIPAS2E1ISNXS	FEAT_XS
100	1001	0000	101	IPAS2LE1ISNXS	FEAT_XS
100	1001	0000	110	RIPAS2LE1ISNXS	FEAT_XS
100	1001	0001	000	ALLE2OSNXS	FEAT_XS
100	1001	0001	001	VAE2OSNXS	FEAT_XS
100	1001	0001	100	ALLE1OSNXS	FEAT_XS
100	1001	0001	101	VALE2OSNXS	FEAT_XS
100	1001	0001	110	VMALLS12E1OSNXS	FEAT_XS
100	1001	0010	001	RVAE2ISNXS	FEAT_XS
100	1001	0010	101	RVALE2ISNXS	FEAT_XS
100	1001	0011	000	ALLE2ISNXS	FEAT_XS
100	1001	0011	001	VAE2ISNXS	FEAT_XS
100	1001	0011	100	ALLE1ISNXS	FEAT_XS
100	1001	0011	101	VALE2ISNXS	FEAT_XS
100	1001	0011	110	VMALLS12E1ISNXS	FEAT_XS
100	1001	0100	000	IPAS2E1OSNXS	FEAT_XS
100	1001	0100	001	IPAS2E1NXS	FEAT_XS
100	1001	0100	010	RIPAS2E1NXS	FEAT_XS
100	1001	0100	011	RIPAS2E1OSNXS	FEAT_XS
100	1001	0100	100	IPAS2LE1OSNXS	FEAT_XS
100	1001	0100	101	IPAS2LE1NXS	FEAT_XS
100	1001	0100	110	RIPAS2LE1NXS	FEAT_XS
100	1001	0100	111	RIPAS2LE1OSNXS	FEAT_XS
100	1001	0101	001	RVAE2OSNXS	FEAT_XS
100	1001	0101	101	RVALE2OSNXS	FEAT_XS
100	1001	0110	001	RVAE2NXS	FEAT_XS
100	1001	0110	101	RVALE2NXS	FEAT_XS
100	1001	0111	000	ALLE2NXS	FEAT_XS
100	1001	0111	001	VAE2NXS	FEAT_XS
100	1001	0111	100	ALLE1NXS	FEAT_XS
100	1001	0111	101	VALE2NXS	FEAT_XS
100	1001	0111	110	VMALLS12E1NXS	FEAT_XS
110	1000	0001	000	ALLE3OS	FEAT_TLBIOS
110	1000	0001	001	VAE3OS	FEAT_TLBIOS
110	1000	0001	101	VALE3OS	FEAT_TLBIOS
110	1000	0010	001	RVAE3IS	FEAT_TLBIRANGE
110	1000	0010	101	RVALE3IS	FEAT_TLBIRANGE
110	1000	0011	000	ALLE3IS	-
110	1000	0011	001	VAE3IS	-
110	1000	0011	101	VALE3IS	-
110	1000	0101	001	RVAE3OS	FEAT_TLBIRANGE
110	1000	0101	101	RVALE3OS	FEAT_TLBIRANGE

op1	CRn	CRm	op2	<tlbi_op>	Architectural Feature
110	1000	0110	001	RVAE3	FEAT_TLBIRANGE
110	1000	0110	101	RVALE3	FEAT_TLBIRANGE
110	1000	0111	000	ALLE3	-
110	1000	0111	001	VAE3	-
110	1000	0111	101	VALE3	-
110	1001	0001	000	ALLE3OSNXS	FEAT_XS
110	1001	0001	001	VAE3OSNXS	FEAT_XS
110	1001	0001	101	VALE3OSNXS	FEAT_XS
110	1001	0010	001	RVAE3ISNXS	FEAT_XS
110	1001	0010	101	RVALE3ISNXS	FEAT_XS
110	1001	0011	000	ALLE3ISNXS	FEAT_XS
110	1001	0011	001	VAE3ISNXS	FEAT_XS
110	1001	0011	101	VALE3ISNXS	FEAT_XS
110	1001	0101	001	RVAE3OSNXS	FEAT_XS
110	1001	0101	101	RVALE3OSNXS	FEAT_XS
110	1001	0110	001	RVAE3NXS	FEAT_XS
110	1001	0110	101	RVALE3NXS	FEAT_XS
110	1001	0111	000	ALLE3NXS	FEAT_XS
110	1001	0111	001	VAE3NXS	FEAT_XS
110	1001	0111	101	VALE3NXS	FEAT_XS

<Xt> Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

Operation

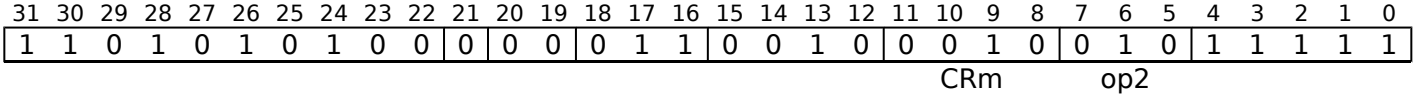
The description of SYS gives the operational pseudocode for this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

TSB CSYNC

Trace Synchronization Barrier. This instruction is a barrier that synchronizes the trace operations of instructions. If *FEAT_TRF* is not implemented, this instruction executes as a NOP.

System
(FEAT_TRF)



TSB CSYNC

```
if !HaveSelfHostedTrace() then EndOfInstruction();
```

Operation

```
TraceSynchronizationBarrier();
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

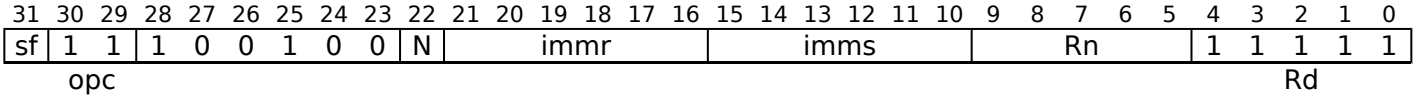
TST (immediate)

Test bits (immediate), setting the condition flags and discarding the result

: `Rn AND imm`.

This is an alias of ANDS (immediate). This means:

- The encodings in this description are named to match the encodings of ANDS (immediate).
- The description of ANDS (immediate) gives the operational pseudocode for this instruction.



32-bit (sf == 0 && N == 0)

```
TST <Wn>, #<imm>
```

is equivalent to

```
ANDS WZR, <Wn>, #<imm>
```

and is always the preferred disassembly.

64-bit (sf == 1)

```
TST <Xn>, #<imm>
```

is equivalent to

```
ANDS XZR, <Xn>, #<imm>
```

and is always the preferred disassembly.

Assembler Symbols

<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<imm>	For the 32-bit variant: is the bitmask immediate, encoded in "imms:immr". For the 64-bit variant: is the bitmask immediate, encoded in "N:imms:immr".

Operation

The description of ANDS (immediate) gives the operational pseudocode for this instruction.

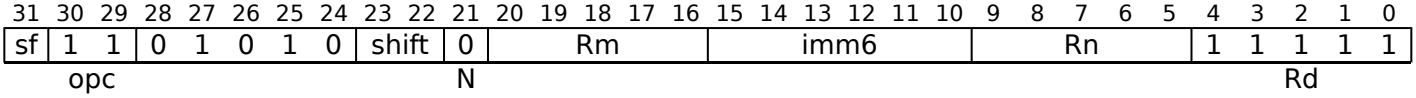
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

TST (shifted register)

Test (shifted register) performs a bitwise AND operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

This is an alias of ANDS (shifted register). This means:

- The encodings in this description are named to match the encodings of ANDS (shifted register).
- The description of ANDS (shifted register) gives the operational pseudocode for this instruction.



32-bit (sf == 0)

```
TST <Wn>, <Wm>{, <shift> #<amount>}
```

is equivalent to

```
ANDS WZR, <Wn>, <Wm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

64-bit (sf == 1)

```
TST <Xn>, <Xm>{, <shift> #<amount>}
```

is equivalent to

```
ANDS XZR, <Xn>, <Xm>{, <shift> #<amount>}
```

and is always the preferred disassembly.

Assembler Symbols

<Wn> Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Wm> Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.

<Xn> Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

<shift> Is the optional shift to be applied to the final source, defaulting to LSL and encoded in “shift”:

shift	<shift>
00	LSL
01	LSR
10	ASR
11	ROR

<amount> For the 32-bit variant: is the shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm6" field.

For the 64-bit variant: is the shift amount, in the range 0 to 63, defaulting to 0 and encoded in the "imm6" field,

Operation

The description of ANDS (shifted register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

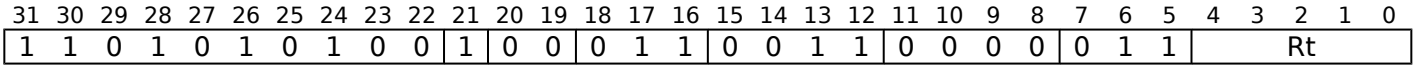
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

TSTART

This instruction starts a new transaction. If the transaction started successfully, the destination register is set to zero. If the transaction failed or was canceled, then all state modifications that were performed transactionally are discarded and the destination register is written with a non-zero value that encodes the cause of the failure.

System
(FEAT_TME)



TSTART <Xt>

```
if !HaveTME() then UNDEFINED;
integer t = UInt(Rt);
```

Assembler Symbols

<Xt>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.

Operation

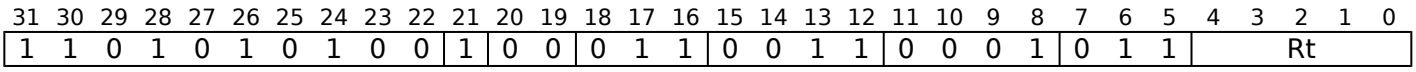
```
CheckTMEEnabled();

boolean IsEL1Regime;
bit tme;
bit tmt;
case PSTATE.EL of
    when EL0
        IsEL1Regime = S1TranslationRegime() == EL1;
        if IsEL1Regime then
            tme = SCTLR_EL1.TME0;
            tmt = SCTLR_EL1.TMT0;
        else
            tme = SCTLR_EL2.TME0;
            tmt = SCTLR_EL2.TMT0;
    when EL1
        tme = SCTLR_EL1.TME;
        tmt = SCTLR_EL1.TMT;
    when EL2
        tme = SCTLR_EL2.TME;
        tmt = SCTLR_EL2.TMT;
    when EL3
        tme = SCTLR_EL3.TME;
        tmt = SCTLR_EL3.TMT;
    otherwise
        Unreachable();

enable = tme == '1';
trivial = tmt == '1';

if !enable then
    TransactionStartTrap(t);
elsif trivial then
    TSTATE.nPC = NextInstrAddr(64);
    TSTATE.Rt = t;
    FailTransaction(TMFailure_TRIVIAL, FALSE);
elsif HaveSME() && PSTATE.SM == '1' then
    FailTransaction(TMFailure_ERR, FALSE);
elsif TSTATE.depth == 255 then
    FailTransaction(TMFailure_NEST, FALSE);
elsif TSTATE.depth == 0 then
    TSTATE.nPC = NextInstrAddr(64);
    TSTATE.Rt = t;
    ClearExclusiveLocal(ProcessorID());
    TakeTransactionCheckpoint();
    StartTrackingTransactionalReadsWrites();

TSTATE.depth = TSTATE.depth + 1;
X[t, 64] = Zeros(64);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

TTEST

This instruction writes the depth of the transaction to the destination register, or the value 0 otherwise.

System
(FEAT_TME)



TTEST <Xt>

```
if !HaveTME() then UNDEFINED;
integer t = UInt(Rt);
```

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.

Operation

```
CheckTMEEnabled();

X[t, 64] = (TSTATE.depth)<63:0>;
```

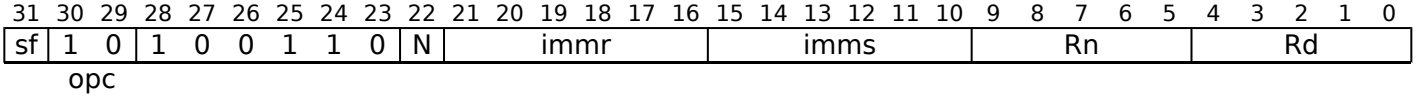
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UBFIZ

Unsigned Bitfield Insert in Zeros copies a bitfield of <width> bits from the least significant bits of the source register to bit position <lsb> of the destination register, setting the destination bits above and below the bitfield to zero.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.



32-bit (sf == 0 && N == 0)

```
UBFIZ <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
UBFM <Wd>, <Wn>, #(-<lsb> MOD 32), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

64-bit (sf == 1 && N == 1)

```
UBFIZ <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
UBFM <Xd>, <Xn>, #(-<lsb> MOD 64), #(<width>-1)
```

and is the preferred disassembly when `UInt(imms) < UInt(immr)`.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<lsb>	For the 32-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 31. For the 64-bit variant: is the bit number of the lsb of the destination bitfield, in the range 0 to 63.
<width>	For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>. For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

UBFM

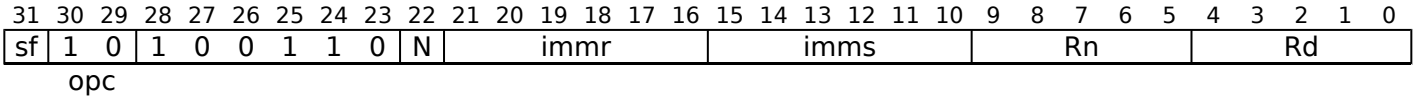
Unsigned Bitfield Move is usually accessed via one of its aliases, which are always preferred for disassembly.

If <imms> is greater than or equal to <immr>, this copies a bitfield of (<imms>-<immr>+1) bits starting from bit position <immr> in the source register to the least significant bits of the destination register.

If <imms> is less than <immr>, this copies a bitfield of (<imms>+1) bits from the least significant bits of the source register to bit position (regsize-<immr>) of the destination register, where regsize is the destination register size of 32 or 64 bits.

In both cases the destination bits below and above the bitfield are set to zero.

This instruction is used by the aliases LSL (immediate), LSR (immediate), UBFIZ, UBFX, UXTB, and UXTH.



32-bit (sf == 0 && N == 0)

```
UBFM <Wd>, <Wn>, #<immr>, #<imms>
```

64-bit (sf == 1 && N == 1)

```
UBFM <Xd>, <Xn>, #<immr>, #<imms>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer datasize = if sf == '1' then 64 else 32;

integer r;
bits(datasize) wmask;
bits(datasize) tmask;

if sf == '1' && N != '1' then UNDEFINED;
if sf == '0' && (N != '0' || immr<5> != '0' || imms<5> != '0') then UNDEFINED;

r = UInt(immr);
(wmask, tmask) = DecodeBitMasks(N, imms, immr, FALSE, datasize);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<immr>	For the 32-bit variant: is the right rotate amount, in the range 0 to 31, encoded in the "immr" field. For the 64-bit variant: is the right rotate amount, in the range 0 to 63, encoded in the "immr" field.
<imms>	For the 32-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 31, encoded in the "imms" field. For the 64-bit variant: is the leftmost bit number to be moved from the source, in the range 0 to 63, encoded in the "imms" field.

Alias Conditions

Alias	Of variant	Is preferred when
LSL (immediate)	32-bit	`imms != '011111' && imms + 1 == immr`
LSL (immediate)	64-bit	`imms != '111111' && imms + 1 == immr`
LSR (immediate)	32-bit	`imms == '011111'`
LSR (immediate)	64-bit	`imms == '111111'`

Alias	Of variant	Is preferred when
UBFIZ		`UInt(imms) < UInt(immr)`
UBFX		`BFXPreferred(sf, opc<1>, imms, immr)`
UXTB		`immr == '000000' && imms == '000111'`
UXTH		`immr == '000000' && imms == '001111'`

Operation

```
bits(datasize) src = X[n, datasize];

// perform bitfield move on low bits
bits(datasize) bot = ROR(src, r) AND wmask;

// combine extension bits and result bits
X[d, datasize] = bot AND tmask;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

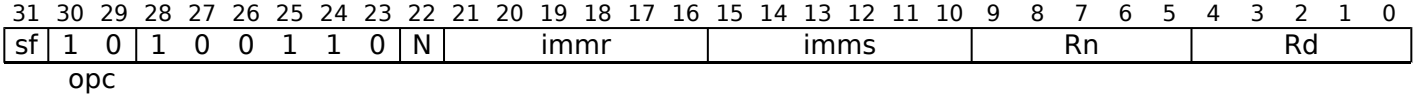
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UBFX

Unsigned Bitfield Extract copies a bitfield of <width> bits starting from bit position <lsb> in the source register to the least significant bits of the destination register, and sets destination bits above the bitfield to zero.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.



32-bit (sf == 0 && N == 0)

```
UBFX <Wd>, <Wn>, #<lsb>, #<width>
```

is equivalent to

```
UBFM <Wd>, <Wn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `BFXPreferred(sf, opc<1>, imms, immr)`.

64-bit (sf == 1 && N == 1)

```
UBFX <Xd>, <Xn>, #<lsb>, #<width>
```

is equivalent to

```
UBFM <Xd>, <Xn>, #<lsb>, #(<lsb>+<width>-1)
```

and is the preferred disassembly when `BFXPreferred(sf, opc<1>, imms, immr)`.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<lsb>	For the 32-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 31. For the 64-bit variant: is the bit number of the lsb of the source bitfield, in the range 0 to 63.
<width>	For the 32-bit variant: is the width of the bitfield, in the range 1 to 32-<lsb>. For the 64-bit variant: is the width of the bitfield, in the range 1 to 64-<lsb>.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

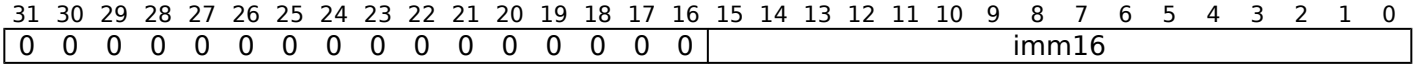
Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

UDF

Permanently Undefined generates an Undefined Instruction exception (ESR_ELx.EC = 0b000000). The encodings for UDF used in this section are defined as permanently UNDEFINED.



UDF #<imm>

```
// The imm16 field is ignored by hardware.
UNDEFINED;
```

Assembler Symbols

<imm>	is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm16" field. The PE ignores the value of this constant.

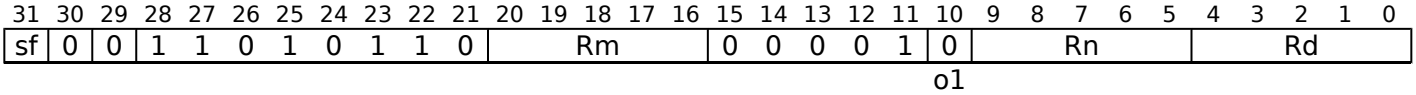
Operation

```
// No operation.
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UDIV

Unsigned Divide divides an unsigned integer register value by another unsigned integer register value, and writes the result to the destination register. The condition flags are not affected.



32-bit (sf == 0)

```
UDIV <Wd>, <Wn>, <Wm>
```

64-bit (sf == 1)

```
UDIV <Xd>, <Xn>, <Xm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if sf == '1' then 64 else 32;
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register, encoded in the "Rm" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register, encoded in the "Rm" field.

Operation

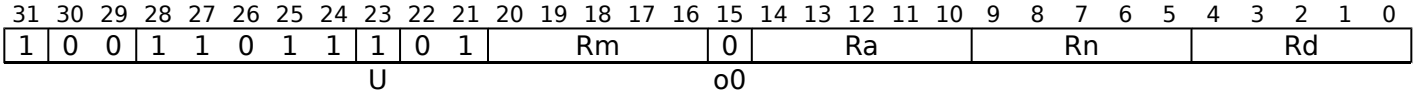
```
bits(datasize) operand1 = X[n, datasize];
bits(datasize) operand2 = X[m, datasize];
integer result;

if IsZero(operand2) then
    result = 0;
else
    result = RoundTowardsZero(Real(Int(operand1, TRUE)) / Real(Int(operand2, TRUE)));

X[d, datasize] = result<datasize-1:0>;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMADDL

Unsigned Multiply-Add Long multiplies two 32-bit register values, adds a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias UMULL.



UMADDL <Xd>, <Wn>, <Wm>, <Xa>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
```

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Xa>	Is the 64-bit name of the third general-purpose source register holding the addend, encoded in the "Ra" field.

Alias Conditions

Alias	Is preferred when
UMULL	`Ra == '11111'`

Operation

```
bits(32) operand1 = X[n, 32];
bits(32) operand2 = X[m, 32];
bits(64) operand3 = X[a, 64];

integer result;

result = Int(operand3, TRUE) + (Int(operand1, TRUE) * Int(operand2, TRUE));

X[d, 64] = result<63:0>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

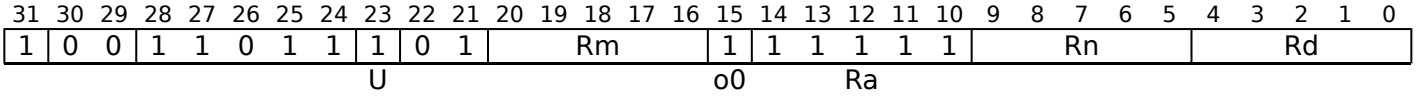
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMNEGL

Unsigned Multiply-Negate Long multiplies two 32-bit register values, negates the product, and writes the result to the 64-bit destination register.

This is an alias of UMSUBL. This means:

- The encodings in this description are named to match the encodings of UMSUBL.
- The description of UMSUBL gives the operational pseudocode for this instruction.



UMNEGL <Xd>, <Wn>, <Wm>

is equivalent to

UMSUBL <Xd>, <Wn>, <Wm>, XZR

and is always the preferred disassembly.

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of UMSUBL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

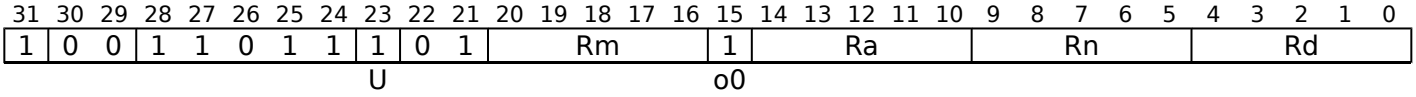
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMSUBL

Unsigned Multiply-Subtract Long multiplies two 32-bit register values, subtracts the product from a 64-bit register value, and writes the result to the 64-bit destination register.

This instruction is used by the alias UMNEGL.



UMSUBL <Xd>, <Wn>, <Wm>, <Xa>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
```

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.
<Xa>	Is the 64-bit name of the third general-purpose source register holding the minuend, encoded in the "Ra" field.

Alias Conditions

Alias	Is preferred when
UMNEGL	`Ra == '11111'`

Operation

```
bits(32) operand1 = X[n, 32];
bits(32) operand2 = X[m, 32];
bits(64) operand3 = X[a, 64];

integer result;

result = Int(operand3, TRUE) - (Int(operand1, TRUE) * Int(operand2, TRUE));
X[d, 64] = result<63:0>;
```

Operational information

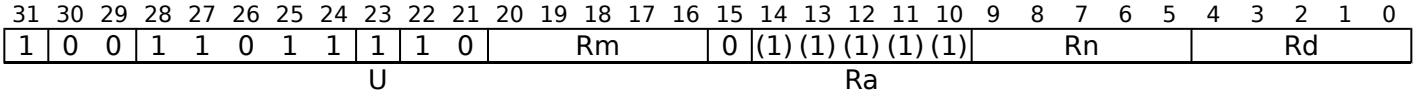
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMULH

Unsigned Multiply High multiplies two 64-bit register values, and writes bits[127:64] of the 128-bit result to the 64-bit destination register.



UMULH <Xd>, <Xn>, <Xm>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

```
bits(64) operand1 = X[n, 64];
bits(64) operand2 = X[m, 64];

integer result;

result = Int(operand1, TRUE) * Int(operand2, TRUE);

X[d, 64] = result<127:64>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

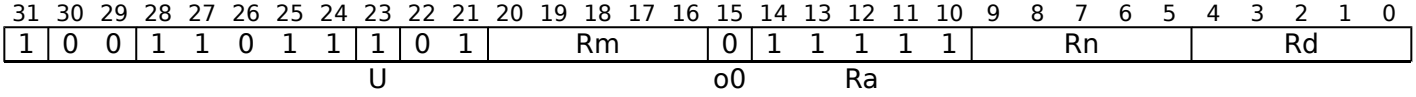
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMULL

Unsigned Multiply Long multiplies two 32-bit register values, and writes the result to the 64-bit destination register.

This is an alias of UMADDL. This means:

- The encodings in this description are named to match the encodings of UMADDL.
- The description of UMADDL gives the operational pseudocode for this instruction.



`UMULL <Xd>, <Wn>, <Wm>`

is equivalent to

`UMADDL <Xd>, <Wn>, <Wm>, XZR`

and is always the preferred disassembly.

Assembler Symbols

<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Wm>	Is the 32-bit name of the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

The description of UMADDL gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

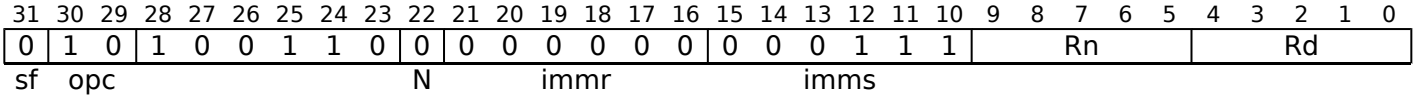
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UXTB

Unsigned Extend Byte extracts an 8-bit value from a register, zero-extends it to the size of the register, and writes the result to the destination register.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.



32-bit

```
UXTB <Wd>, <Wn>
```

is equivalent to

```
UBFM <Wd>, <Wn>, #0, #7
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

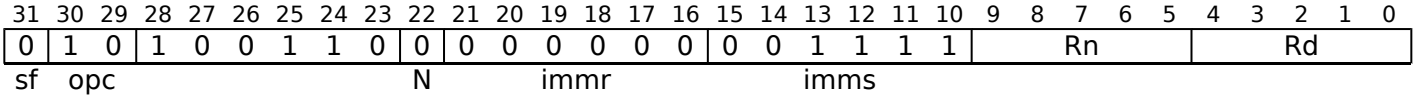
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UXTH

Unsigned Extend Halfword extracts a 16-bit value from a register, zero-extends it to the size of the register, and writes the result to the destination register.

This is an alias of UBFM. This means:

- The encodings in this description are named to match the encodings of UBFM.
- The description of UBFM gives the operational pseudocode for this instruction.



32-bit

```
UXTH <Wd>, <Wn>
```

is equivalent to

```
UBFM <Wd>, <Wn>, #0, #15
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

The description of UBFM gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

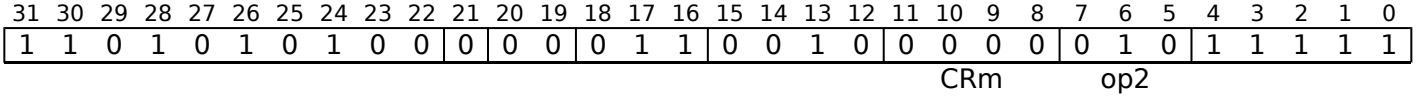
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

WFE

Wait For Event is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. Wakeup events include the event signaled as a result of executing the SEV instruction on any PE in the multiprocessor system. For more information, see *Wait For Event mechanism and Send event*.

As described in *Wait For Event mechanism and Send event*, the execution of a WFE instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level.



WFE

```
// Empty.
```

Operation

```
integer localtimeout = 1 << 64;    // No local timeout event is generated
Hint_WFE(localtimeout, WFxType_WFE);
```

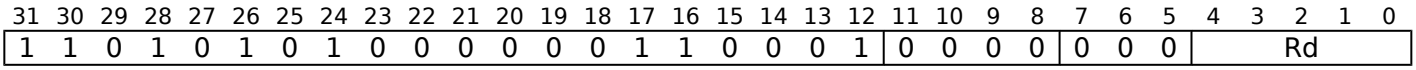
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

WFET

Wait For Event with Timeout is a hint instruction that indicates that the PE can enter a low-power state and remain there until either a local timeout event or a wakeup event occurs. Wakeup events include the event signaled as a result of executing the SEV instruction on any PE in the multiprocessor system. For more information, see *Wait For Event mechanism and Send event*.

As described in *Wait For Event mechanism and Send event*, the execution of a WFET instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level.

System
(FEAT_WFxT)



WFET <Xt>

```
if !HaveFeatWFxT() then UNDEFINED;

integer d = UInt(Rd);
```

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rd" field.

Operation

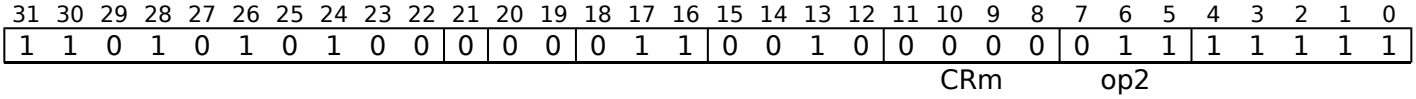
```
integer localtimeout = UInt(X[d, 64]);

if Halted() && ConstrainUnpredictableBool(Unpredictable_WFxTDEBUG) then
    EndOfInstruction();

Hint_WFE(localtimeout, WFxType_WFET);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

WFI

Wait For Interrupt is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. For more information, see *Wait For Interrupt*.

As described in *Wait For Interrupt*, the execution of a `WFI` instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level.



WFI

```
// Empty.
```

Operation

```
integer localtimeout = 1 << 64;    // No local timeout event is generated
Hint_WFI(localtimeout, WFxType_WFI);
```

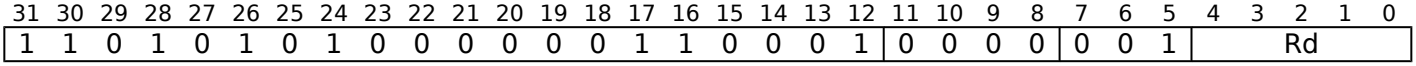
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

WFIT

Wait For Interrupt with Timeout is a hint instruction that indicates that the PE can enter a low-power state and remain there until either a local timeout event or a wakeup event occurs. For more information, see *Wait For Interrupt*.

As described in *Wait For Interrupt*, the execution of a WFIT instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level.

System
(FEAT_WFxT)



WFIT <Xt>

```
if !HaveFeatWFxT() then UNDEFINED;

integer d = UInt(Rd);
```

Assembler Symbols

<Xt> Is the 64-bit name of the general-purpose source register, encoded in the "Rd" field.

Operation

```
integer localtimeout = UInt(X[d, 64]);

if Halted() && ConstrainUnpredictableBool(Unpredictable_WFxTDEBUG) then
    EndOfInstruction();

Hint_WFI(localtimeout, WFxType_WFIT);
```

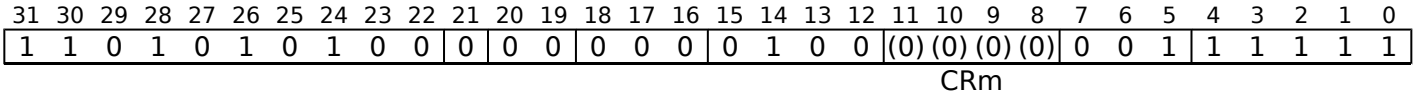
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

XAFLAG

Convert floating-point condition flags from external format to Arm format. This instruction converts the state of the PSTATE.{N,Z,C,V} flags from an alternative representation required by some software to a form representing the result of an Arm floating-point scalar compare instruction.

System

(FEAT_FlagM2)



XAFLAG

```
if !HaveFlagFormatExt() then UNDEFINED;
```

Operation

```
bit n = NOT(PSTATE.C) AND NOT(PSTATE.Z);
bit z = PSTATE.Z AND PSTATE.C;
bit c = PSTATE.C OR PSTATE.Z;
bit v = NOT(PSTATE.C) AND PSTATE.Z;

PSTATE.N = n;
PSTATE.Z = z;
PSTATE.C = c;
PSTATE.V = v;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

XPACD, XPACI, XPACLRI

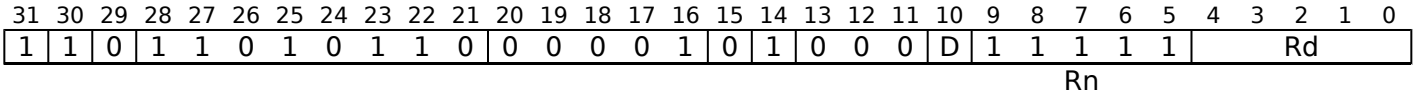
Strip Pointer Authentication Code. This instruction removes the pointer authentication code from an address. The address is in the specified general-purpose register for XPACI and XPACD, and is in LR for XPACLRI.

The XPACD instruction is used for data addresses, and XPACI and XPACLRI are used for instruction addresses.

It has encodings from 2 classes: Integer and System

Integer

(FEAT_PAuth)



XPACD (D == 1)

```
XPACD <Xd>
```

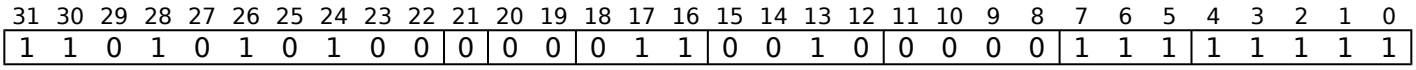
XPACI (D == 0)

```
XPACI <Xd>
```

```
boolean data = (D == '1');
integer d = UInt(Rd);

if !HavePACExt() then
    UNDEFINED;
```

System

(FEAT_PAuth)



XPACLRI

```
integer d = 30;
boolean data = FALSE;
```

Assembler Symbols

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

Operation

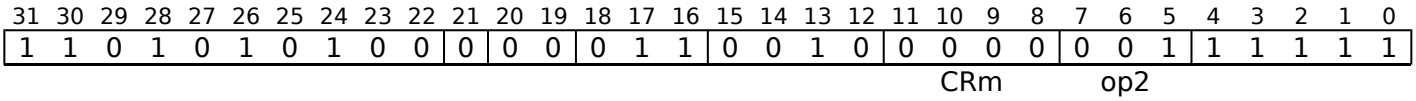
```
if HavePACExt() then
    X[d, 64] = Strip(X[d, 64], data);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

YIELD

YIELD is a hint instruction. Software with a multithreading capability can use a `YIELD` instruction to indicate to the PE that it is performing a task, for example a spin-lock, that could be swapped out to improve overall system performance. The PE can use this hint to suspend and resume multiple software threads if it supports the capability.

For more information about the recommended use of this instruction, see *The YIELD instruction*.



YIELD

```
// Empty.
```

Operation

```
Hint_Yield();
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

A64 -- SIMD and Floating-point Instructions (alphabetic order)

ABS: Absolute value (vector).

ADD (vector): Add (vector).

ADDHN, ADDHN2: Add returning High Narrow.

ADDP (scalar): Add Pair of elements (scalar).

ADDP (vector): Add Pairwise (vector).

ADDV: Add across Vector.

AESD: AES single round decryption.

AESE: AES single round encryption.

AESIMC: AES inverse mix columns.

AESMC: AES mix columns.

AND (vector): Bitwise AND (vector).

BCAX: Bit Clear and XOR.

BFCVT: Floating-point convert from single-precision to BFloat16 format (scalar).

BFCVTN, BFCVTN2: Floating-point convert from single-precision to BFloat16 format (vector).

BFDOT (by element): BFloat16 floating-point dot product (vector, by element).

BFDOT (vector): BFloat16 floating-point dot product (vector).

BFMLALB, BFMLALT (by element): BFloat16 floating-point widening multiply-add long (by element).

BFMLALB, BFMLALT (vector): BFloat16 floating-point widening multiply-add long (vector).

BFMMLA: BFloat16 floating-point matrix multiply-accumulate into 2x2 matrix.

BIC (vector, immediate): Bitwise bit Clear (vector, immediate).

BIC (vector, register): Bitwise bit Clear (vector, register).

BIF: Bitwise Insert if False.

BIT: Bitwise Insert if True.

BSL: Bitwise Select.

CLS (vector): Count Leading Sign bits (vector).

CLZ (vector): Count Leading Zero bits (vector).

CMEQ (register): Compare bitwise Equal (vector).

CMEQ (zero): Compare bitwise Equal to zero (vector).

CMGE (register): Compare signed Greater than or Equal (vector).

CMGE (zero): Compare signed Greater than or Equal to zero (vector).

CMGT (register): Compare signed Greater than (vector).

CMGT (zero): Compare signed Greater than zero (vector).

CMHI (register): Compare unsigned Higher (vector).

CMHS (register): Compare unsigned Higher or Same (vector).

CMLE (zero): Compare signed Less than or Equal to zero (vector).

CMLT (zero): Compare signed Less than zero (vector).

CMTST: Compare bitwise Test bits nonzero (vector).

CNT: Population Count per byte.

DUP (element): Duplicate vector element to vector or scalar.

DUP (general): Duplicate general-purpose register to vector.

EOR (vector): Bitwise Exclusive OR (vector).

EOR3: Three-way Exclusive OR.

EXT: Extract vector from pair of vectors.

FABD: Floating-point Absolute Difference (vector).

FABS (scalar): Floating-point Absolute value (scalar).

FABS (vector): Floating-point Absolute value (vector).

FACGE: Floating-point Absolute Compare Greater than or Equal (vector).

FACGT: Floating-point Absolute Compare Greater than (vector).

FADD (scalar): Floating-point Add (scalar).

FADD (vector): Floating-point Add (vector).

FADDP (scalar): Floating-point Add Pair of elements (scalar).

FADDP (vector): Floating-point Add Pairwise (vector).

FCADD: Floating-point Complex Add.

FCCMP: Floating-point Conditional quiet Compare (scalar).

FCCMPE: Floating-point Conditional signaling Compare (scalar).

FCMEQ (register): Floating-point Compare Equal (vector).

FCMEQ (zero): Floating-point Compare Equal to zero (vector).

FCMGE (register): Floating-point Compare Greater than or Equal (vector).

FCMGE (zero): Floating-point Compare Greater than or Equal to zero (vector).

FCMGT (register): Floating-point Compare Greater than (vector).

FCMGT (zero): Floating-point Compare Greater than zero (vector).

FCMLA: Floating-point Complex Multiply Accumulate.

FCMLA (by element): Floating-point Complex Multiply Accumulate (by element).

FCMLE (zero): Floating-point Compare Less than or Equal to zero (vector).

FCMLT (zero): Floating-point Compare Less than zero (vector).

FCMP: Floating-point quiet Compare (scalar).

FCMPE: Floating-point signaling Compare (scalar).

FCSEL: Floating-point Conditional Select (scalar).

FCVT: Floating-point Convert precision (scalar).

FCVTAS (scalar): Floating-point Convert to Signed integer, rounding to nearest with ties to Away (scalar).

FCVTAS (vector): Floating-point Convert to Signed integer, rounding to nearest with ties to Away (vector).

FCVTAU (scalar): Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (scalar).

FCVTAU (vector): Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (vector).

FCVTL, FCVTL2: Floating-point Convert to higher precision Long (vector).

FCVTMS (scalar): Floating-point Convert to Signed integer, rounding toward Minus infinity (scalar).

FCVTMS (vector): Floating-point Convert to Signed integer, rounding toward Minus infinity (vector).

FCVTMU (scalar): Floating-point Convert to Unsigned integer, rounding toward Minus infinity (scalar).

FCVTMU (vector): Floating-point Convert to Unsigned integer, rounding toward Minus infinity (vector).

FCVTN, FCVTN2: Floating-point Convert to lower precision Narrow (vector).

FCVTNS (scalar): Floating-point Convert to Signed integer, rounding to nearest with ties to even (scalar).

FCVTNS (vector): Floating-point Convert to Signed integer, rounding to nearest with ties to even (vector).

FCVTNU (scalar): Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (scalar).

FCVTNU (vector): Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (vector).

FCVTPS (scalar): Floating-point Convert to Signed integer, rounding toward Plus infinity (scalar).

FCVTPS (vector): Floating-point Convert to Signed integer, rounding toward Plus infinity (vector).

FCVTPU (scalar): Floating-point Convert to Unsigned integer, rounding toward Plus infinity (scalar).

FCVTPU (vector): Floating-point Convert to Unsigned integer, rounding toward Plus infinity (vector).

FCVTXN, FCVTXN2: Floating-point Convert to lower precision Narrow, rounding to odd (vector).

FCVTZS (scalar, fixed-point): Floating-point Convert to Signed fixed-point, rounding toward Zero (scalar).

FCVTZS (scalar, integer): Floating-point Convert to Signed integer, rounding toward Zero (scalar).

FCVTZS (vector, fixed-point): Floating-point Convert to Signed fixed-point, rounding toward Zero (vector).

FCVTZS (vector, integer): Floating-point Convert to Signed integer, rounding toward Zero (vector).

FCVTZU (scalar, fixed-point): Floating-point Convert to Unsigned fixed-point, rounding toward Zero (scalar).

FCVTZU (scalar, integer): Floating-point Convert to Unsigned integer, rounding toward Zero (scalar).

FCVTZU (vector, fixed-point): Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector).

FCVTZU (vector, integer): Floating-point Convert to Unsigned integer, rounding toward Zero (vector).

FDIV (scalar): Floating-point Divide (scalar).

FDIV (vector): Floating-point Divide (vector).

FJCVTZS: Floating-point Javascript Convert to Signed fixed-point, rounding toward Zero.

FMADD: Floating-point fused Multiply-Add (scalar).

FMAX (scalar): Floating-point Maximum (scalar).

FMAX (vector): Floating-point Maximum (vector).

FMAXNM (scalar): Floating-point Maximum Number (scalar).

FMAXNM (vector): Floating-point Maximum Number (vector).

FMAXNMP (scalar): Floating-point Maximum Number of Pair of elements (scalar).

FMAXNMP (vector): Floating-point Maximum Number Pairwise (vector).

FMAXNMV: Floating-point Maximum Number across Vector.

FMAXP (scalar): Floating-point Maximum of Pair of elements (scalar).

FMAXP (vector): Floating-point Maximum Pairwise (vector).

FMAXV: Floating-point Maximum across Vector.

FMIN (scalar): Floating-point Minimum (scalar).

FMIN (vector): Floating-point minimum (vector).

FMINNM (scalar): Floating-point Minimum Number (scalar).

FMINNM (vector): Floating-point Minimum Number (vector).

FMINNMP (scalar): Floating-point Minimum Number of Pair of elements (scalar).

FMINNMP (vector): Floating-point Minimum Number Pairwise (vector).

FMINNMV: Floating-point Minimum Number across Vector.

FMINP (scalar): Floating-point Minimum of Pair of elements (scalar).

FMINP (vector): Floating-point Minimum Pairwise (vector).

FMINV: Floating-point Minimum across Vector.

FMLA (by element): Floating-point fused Multiply-Add to accumulator (by element).

FMLA (vector): Floating-point fused Multiply-Add to accumulator (vector).

FMLAL, FMLAL2 (by element): Floating-point fused Multiply-Add Long to accumulator (by element).

FMLAL, FMLAL2 (vector): Floating-point fused Multiply-Add Long to accumulator (vector).

FMLS (by element): Floating-point fused Multiply-Subtract from accumulator (by element).

FMLS (vector): Floating-point fused Multiply-Subtract from accumulator (vector).

FMLSL, FMLSL2 (by element): Floating-point fused Multiply-Subtract Long from accumulator (by element).

FMLSL, FMLSL2 (vector): Floating-point fused Multiply-Subtract Long from accumulator (vector).

FMOV (general): Floating-point Move to or from general-purpose register without conversion.

FMOV (register): Floating-point Move register without conversion.

FMOV (scalar, immediate): Floating-point move immediate (scalar).

FMOV (vector, immediate): Floating-point move immediate (vector).

FMSUB: Floating-point Fused Multiply-Subtract (scalar).

FMUL (by element): Floating-point Multiply (by element).

FMUL (scalar): Floating-point Multiply (scalar).

FMUL (vector): Floating-point Multiply (vector).

FMULX: Floating-point Multiply extended.

FMULX (by element): Floating-point Multiply extended (by element).

FNEG (scalar): Floating-point Negate (scalar).

FNEG (vector): Floating-point Negate (vector).

FNMADD: Floating-point Negated fused Multiply-Add (scalar).

FNMSUB: Floating-point Negated fused Multiply-Subtract (scalar).

FNMUL (scalar): Floating-point Multiply-Negate (scalar).

FRECPE: Floating-point Reciprocal Estimate.

FRECPS: Floating-point Reciprocal Step.

FRECPX: Floating-point Reciprocal exponent (scalar).

FRINT32X (scalar): Floating-point Round to 32-bit Integer, using current rounding mode (scalar).

FRINT32X (vector): Floating-point Round to 32-bit Integer, using current rounding mode (vector).

FRINT32Z (scalar): Floating-point Round to 32-bit Integer toward Zero (scalar).

FRINT32Z (vector): Floating-point Round to 32-bit Integer toward Zero (vector).

FRINT64X (scalar): Floating-point Round to 64-bit Integer, using current rounding mode (scalar).

FRINT64X (vector): Floating-point Round to 64-bit Integer, using current rounding mode (vector).

FRINT64Z (scalar): Floating-point Round to 64-bit Integer toward Zero (scalar).

FRINT64Z (vector): Floating-point Round to 64-bit Integer toward Zero (vector).

FRINTA (scalar): Floating-point Round to Integral, to nearest with ties to Away (scalar).

FRINTA (vector): Floating-point Round to Integral, to nearest with ties to Away (vector).

FRINTI (scalar): Floating-point Round to Integral, using current rounding mode (scalar).

FRINTI (vector): Floating-point Round to Integral, using current rounding mode (vector).

FRINTM (scalar): Floating-point Round to Integral, toward Minus infinity (scalar).

FRINTM (vector): Floating-point Round to Integral, toward Minus infinity (vector).

FRINTN (scalar): Floating-point Round to Integral, to nearest with ties to even (scalar).

FRINTN (vector): Floating-point Round to Integral, to nearest with ties to even (vector).

FRINTP (scalar): Floating-point Round to Integral, toward Plus infinity (scalar).

FRINTP (vector): Floating-point Round to Integral, toward Plus infinity (vector).

FRINTX (scalar): Floating-point Round to Integral exact, using current rounding mode (scalar).

FRINTX (vector): Floating-point Round to Integral exact, using current rounding mode (vector).

FRINTZ (scalar): Floating-point Round to Integral, toward Zero (scalar).

FRINTZ (vector): Floating-point Round to Integral, toward Zero (vector).

FRSQRTE: Floating-point Reciprocal Square Root Estimate.

FRSQRTS: Floating-point Reciprocal Square Root Step.

FSQRT (scalar): Floating-point Square Root (scalar).

FSQRT (vector): Floating-point Square Root (vector).

FSUB (scalar): Floating-point Subtract (scalar).

FSUB (vector): Floating-point Subtract (vector).

INS (element): Insert vector element from another vector element.

INS (general): Insert vector element from general-purpose register.

LD1 (multiple structures): Load multiple single-element structures to one, two, three, or four registers.

LD1 (single structure): Load one single-element structure to one lane of one register.

LD1R: Load one single-element structure and Replicate to all lanes (of one register).

LD2 (multiple structures): Load multiple 2-element structures to two registers.

LD2 (single structure): Load single 2-element structure to one lane of two registers.

LD2R: Load single 2-element structure and Replicate to all lanes of two registers.

LD3 (multiple structures): Load multiple 3-element structures to three registers.

LD3 (single structure): Load single 3-element structure to one lane of three registers.

LD3R: Load single 3-element structure and Replicate to all lanes of three registers.

LD4 (multiple structures): Load multiple 4-element structures to four registers.

LD4 (single structure): Load single 4-element structure to one lane of four registers.

LD4R: Load single 4-element structure and Replicate to all lanes of four registers.

LDNP (SIMD&FP): Load Pair of SIMD&FP registers, with Non-temporal hint.

LDP (SIMD&FP): Load Pair of SIMD&FP registers.

LDR (immediate, SIMD&FP): Load SIMD&FP Register (immediate offset).

LDR (literal, SIMD&FP): Load SIMD&FP Register (PC-relative literal).

LDR (register, SIMD&FP): Load SIMD&FP Register (register offset).

LDUR (SIMD&FP): Load SIMD&FP Register (unscaled offset).

MLA (by element): Multiply-Add to accumulator (vector, by element).

MLA (vector): Multiply-Add to accumulator (vector).

MLS (by element): Multiply-Subtract from accumulator (vector, by element).

MLS (vector): Multiply-Subtract from accumulator (vector).

MOV (element): Move vector element to another vector element: an alias of INS (element).

MOV (from general): Move general-purpose register to a vector element: an alias of INS (general).

MOV (scalar): Move vector element to scalar: an alias of DUP (element).

MOV (to general): Move vector element to general-purpose register: an alias of UMOV.

MOV (vector): Move vector: an alias of ORR (vector, register).

MOVI: Move Immediate (vector).

MUL (by element): Multiply (vector, by element).

MUL (vector): Multiply (vector).

MVN: Bitwise NOT (vector): an alias of NOT.

MVNI: Move inverted Immediate (vector).

NEG (vector): Negate (vector).

NOT: Bitwise NOT (vector).

ORN (vector): Bitwise inclusive OR NOT (vector).

ORR (vector, immediate): Bitwise inclusive OR (vector, immediate).

ORR (vector, register): Bitwise inclusive OR (vector, register).

PMUL: Polynomial Multiply.

PMULL, PMULL2: Polynomial Multiply Long.

RADDHN, RADDHN2: Rounding Add returning High Narrow.

RAX1: Rotate and Exclusive OR.

RBIT (vector): Reverse Bit order (vector).

REV16 (vector): Reverse elements in 16-bit halfwords (vector).

REV32 (vector): Reverse elements in 32-bit words (vector).

REV64: Reverse elements in 64-bit doublewords (vector).

RSHRN, RSHRN2: Rounding Shift Right Narrow (immediate).

RSUBHN, RSUBHN2: Rounding Subtract returning High Narrow.

SABA: Signed Absolute difference and Accumulate.

SABAL, SABAL2: Signed Absolute difference and Accumulate Long.

SABD: Signed Absolute Difference.

SABDL, SABDL2: Signed Absolute Difference Long.

SADALP: Signed Add and Accumulate Long Pairwise.

SADDL, SADDL2: Signed Add Long (vector).

SADDLP: Signed Add Long Pairwise.

SADDLV: Signed Add Long across Vector.

SADDW, SADDW2: Signed Add Wide.

SCVTF (scalar, fixed-point): Signed fixed-point Convert to Floating-point (scalar).

SCVTF (scalar, integer): Signed integer Convert to Floating-point (scalar).

SCVTF (vector, fixed-point): Signed fixed-point Convert to Floating-point (vector).

SCVTF (vector, integer): Signed integer Convert to Floating-point (vector).

SDOT (by element): Dot Product signed arithmetic (vector, by element).

SDOT (vector): Dot Product signed arithmetic (vector).

SHA1C: SHA1 hash update (choose).

SHA1H: SHA1 fixed rotate.

SHA1M: SHA1 hash update (majority).

SHA1P: SHA1 hash update (parity).

SHA1SU0: SHA1 schedule update 0.

SHA1SU1: SHA1 schedule update 1.

SHA256H: SHA256 hash update (part 1).

SHA256H2: SHA256 hash update (part 2).

SHA256SU0: SHA256 schedule update 0.

SHA256SU1: SHA256 schedule update 1.

SHA512H: SHA512 Hash update part 1.

SHA512H2: SHA512 Hash update part 2.

SHA512SU0: SHA512 Schedule Update 0.

SHA512SU1: SHA512 Schedule Update 1.

SHADD: Signed Halving Add.

SHL: Shift Left (immediate).

SHLL, SHLL2: Shift Left Long (by element size).

SHRN, SHRN2: Shift Right Narrow (immediate).

SHSUB: Signed Halving Subtract.

SLI: Shift Left and Insert (immediate).

SM3PARTW1: SM3PARTW1.

SM3PARTW2: SM3PARTW2.

SM3SS1: SM3SS1.

SM3TT1A: SM3TT1A.

SM3TT1B: SM3TT1B.

SM3TT2A: SM3TT2A.

SM3TT2B: SM3TT2B.

SM4E: SM4 Encode.

SM4EKEY: SM4 Key.

SMAX: Signed Maximum (vector).

SMAXP: Signed Maximum Pairwise.

SMAXV: Signed Maximum across Vector.

SMIN: Signed Minimum (vector).

SMINP: Signed Minimum Pairwise.

SMINV: Signed Minimum across Vector.

SMLAL, SMLAL2 (by element): Signed Multiply-Add Long (vector, by element).

SMLAL, SMLAL2 (vector): Signed Multiply-Add Long (vector).

SMLSL, SMLSL2 (by element): Signed Multiply-Subtract Long (vector, by element).

SMLSL, SMLSL2 (vector): Signed Multiply-Subtract Long (vector).

SMMLA (vector): Signed 8-bit integer matrix multiply-accumulate (vector).

SMOV: Signed Move vector element to general-purpose register.

SMULL, SMULL2 (by element): Signed Multiply Long (vector, by element).

SMULL, SMULL2 (vector): Signed Multiply Long (vector).

SQABS: Signed saturating Absolute value.

SQADD: Signed saturating Add.

SQDMLAL, SQDMLAL2 (by element): Signed saturating Doubling Multiply-Add Long (by element).

SQDMLAL, SQDMLAL2 (vector): Signed saturating Doubling Multiply-Add Long.

SQDMLSL, SQDMLSL2 (by element): Signed saturating Doubling Multiply-Subtract Long (by element).

SQDMLSL, SQDMLSL2 (vector): Signed saturating Doubling Multiply-Subtract Long.

SQDMULH (by element): Signed saturating Doubling Multiply returning High half (by element).

SQDMULH (vector): Signed saturating Doubling Multiply returning High half.

SQDMULL, SQDMULL2 (by element): Signed saturating Doubling Multiply Long (by element).

SQDMULL, SQDMULL2 (vector): Signed saturating Doubling Multiply Long.

SQNEG: Signed saturating Negate.

SQRDMLAH (by element): Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by element).

SQRDMLAH (vector): Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector).

SQRDMLSH (by element): Signed Saturating Rounding Doubling Multiply Subtract returning High Half (by element).

SQRDMLSH (vector): Signed Saturating Rounding Doubling Multiply Subtract returning High Half (vector).

SQRDMULH (by element): Signed saturating Rounding Doubling Multiply returning High half (by element).

SQRDMULH (vector): Signed saturating Rounding Doubling Multiply returning High half.

SQRSHL: Signed saturating Rounding Shift Left (register).

SQRSHRN, SQRSHRN2: Signed saturating Rounded Shift Right Narrow (immediate).

SQRSHRUN, SQRSHRUN2: Signed saturating Rounded Shift Right Unsigned Narrow (immediate).

SQSHL (immediate): Signed saturating Shift Left (immediate).

SQSHL (register): Signed saturating Shift Left (register).

SQSHLU: Signed saturating Shift Left Unsigned (immediate).

SQSHRN, SQSHRN2: Signed saturating Shift Right Narrow (immediate).

SQSHRUN, SQSHRUN2: Signed saturating Shift Right Unsigned Narrow (immediate).

SQSUB: Signed saturating Subtract.

SQXTN, SQXTN2: Signed saturating extract Narrow.

SQXTUN, SQXTUN2: Signed saturating extract Unsigned Narrow.

SRHADD: Signed Rounding Halving Add.

SRI: Shift Right and Insert (immediate).

SRSHL: Signed Rounding Shift Left (register).

SRSHR: Signed Rounding Shift Right (immediate).

SRSRA: Signed Rounding Shift Right and Accumulate (immediate).

SSHL: Signed Shift Left (register).

SSHLL, SSHLL2: Signed Shift Left Long (immediate).

SSHR: Signed Shift Right (immediate).

SSRA: Signed Shift Right and Accumulate (immediate).

SSUBL, SSUBL2: Signed Subtract Long.

SSUBW, SSUBW2: Signed Subtract Wide.

ST1 (multiple structures): Store multiple single-element structures from one, two, three, or four registers.

ST1 (single structure): Store a single-element structure from one lane of one register.

ST2 (multiple structures): Store multiple 2-element structures from two registers.

ST2 (single structure): Store single 2-element structure from one lane of two registers.

ST3 (multiple structures): Store multiple 3-element structures from three registers.

ST3 (single structure): Store single 3-element structure from one lane of three registers.

ST4 (multiple structures): Store multiple 4-element structures from four registers.

ST4 (single structure): Store single 4-element structure from one lane of four registers.

STNP (SIMD&FP): Store Pair of SIMD&FP registers, with Non-temporal hint.

STP (SIMD&FP): Store Pair of SIMD&FP registers.

STR (immediate, SIMD&FP): Store SIMD&FP register (immediate offset).

STR (register, SIMD&FP): Store SIMD&FP register (register offset).

STUR (SIMD&FP): Store SIMD&FP register (unscaled offset).

SUB (vector): Subtract (vector).

SUBHN, SUBHN2: Subtract returning High Narrow.

SUDOT (by element): Dot product with signed and unsigned integers (vector, by element).

SUQADD: Signed saturating Accumulate of Unsigned value.

SXTL, SXTL2: Signed extend Long: an alias of SSHLL, SSHLL2.

TBL: Table vector Lookup.

TBX: Table vector lookup extension.

TRN1: Transpose vectors (primary).

TRN2: Transpose vectors (secondary).

UABA: Unsigned Absolute difference and Accumulate.

UABAL, UABAL2: Unsigned Absolute difference and Accumulate Long.

UABD: Unsigned Absolute Difference (vector).

UABDL, UABDL2: Unsigned Absolute Difference Long.

UADALP: Unsigned Add and Accumulate Long Pairwise.

UADDL, UADDL2: Unsigned Add Long (vector).

UADDLP: Unsigned Add Long Pairwise.

UADDLV: Unsigned sum Long across Vector.

UADDW, UADDW2: Unsigned Add Wide.

UCVTF (scalar, fixed-point): Unsigned fixed-point Convert to Floating-point (scalar).

UCVTF (scalar, integer): Unsigned integer Convert to Floating-point (scalar).

UCVTF (vector, fixed-point): Unsigned fixed-point Convert to Floating-point (vector).

UCVTF (vector, integer): Unsigned integer Convert to Floating-point (vector).

UDOT (by element): Dot Product unsigned arithmetic (vector, by element).

UDOT (vector): Dot Product unsigned arithmetic (vector).

UHADD: Unsigned Halving Add.

UHSUB: Unsigned Halving Subtract.

UMAX: Unsigned Maximum (vector).

UMAXP: Unsigned Maximum Pairwise.

UMAXV: Unsigned Maximum across Vector.

UMIN: Unsigned Minimum (vector).

UMINP: Unsigned Minimum Pairwise.

UMINV: Unsigned Minimum across Vector.

UMLAL, UMLAL2 (by element): Unsigned Multiply-Add Long (vector, by element).

UMLAL, UMLAL2 (vector): Unsigned Multiply-Add Long (vector).

UMLSL, UMLSL2 (by element): Unsigned Multiply-Subtract Long (vector, by element).

UMLSL, UMLSL2 (vector): Unsigned Multiply-Subtract Long (vector).

UMMLA (vector): Unsigned 8-bit integer matrix multiply-accumulate (vector).

UMOV: Unsigned Move vector element to general-purpose register.

UMULL, UMULL2 (by element): Unsigned Multiply Long (vector, by element).

UMULL, UMULL2 (vector): Unsigned Multiply long (vector).

UQADD: Unsigned saturating Add.

UQRSHL: Unsigned saturating Rounding Shift Left (register).

UQRSHRN, UQRSHRN2: Unsigned saturating Rounded Shift Right Narrow (immediate).

UQSHL (immediate): Unsigned saturating Shift Left (immediate).

UQSHL (register): Unsigned saturating Shift Left (register).

UQSHRN, UQSHRN2: Unsigned saturating Shift Right Narrow (immediate).

UQSUB: Unsigned saturating Subtract.

UQXTN, UQXTN2: Unsigned saturating extract Narrow.

URECPE: Unsigned Reciprocal Estimate.

URHADD: Unsigned Rounding Halving Add.

URSHL: Unsigned Rounding Shift Left (register).

URSHR: Unsigned Rounding Shift Right (immediate).

URSQRTE: Unsigned Reciprocal Square Root Estimate.

URSRA: Unsigned Rounding Shift Right and Accumulate (immediate).

USDOT (by element): Dot Product with unsigned and signed integers (vector, by element).

USDOT (vector): Dot Product with unsigned and signed integers (vector).

USHL: Unsigned Shift Left (register).

USHLL, USHLL2: Unsigned Shift Left Long (immediate).

USHR: Unsigned Shift Right (immediate).

USMMLA (vector): Unsigned and signed 8-bit integer matrix multiply-accumulate (vector).

USQADD: Unsigned saturating Accumulate of Signed value.

USRA: Unsigned Shift Right and Accumulate (immediate).

USUBL, USUBL2: Unsigned Subtract Long.

USUBW, USUBW2: Unsigned Subtract Wide.

UXTL, UXTL2: Unsigned extend Long: an alias of USHLL, USHLL2.

UZP1: Unzip vectors (primary).

UZP2: Unzip vectors (secondary).

XAR: Exclusive OR and Rotate.

XTN, XTN2: Extract Narrow.

ZIP1: Zip vectors (primary).

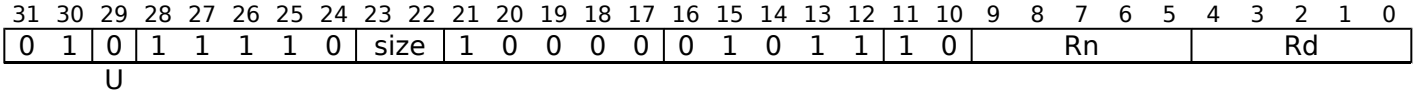
ZIP2: Zip vectors (secondary).

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ABS

Absolute value (vector). This instruction calculates the absolute value of each vector element in the source SIMD&FP register, puts the result into a vector, and writes the vector to the destination SIMD&FP register.
Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.
It has encodings from 2 classes: Scalar and Vector

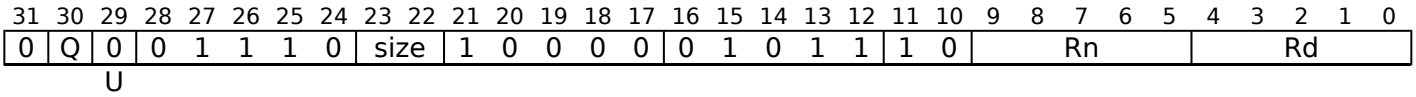
Scalar



ABS <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');
```

Vector



ABS <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

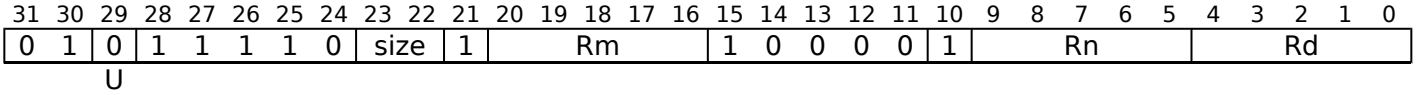
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ADD (vector)

Add (vector). This instruction adds corresponding elements in the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register.
Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.
It has encodings from 2 classes: Scalar and Vector

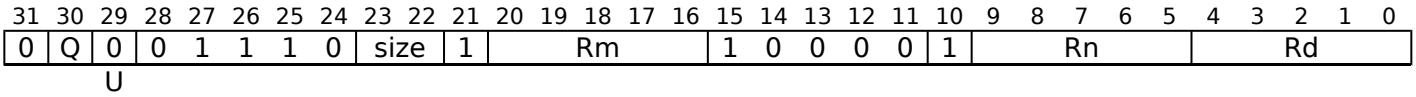
Scalar



ADD <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (U == '1');
```

Vector



ADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then
        Elem[result, e, esize] = element1 - element2;
    else
        Elem[result, e, esize] = element1 + element2;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

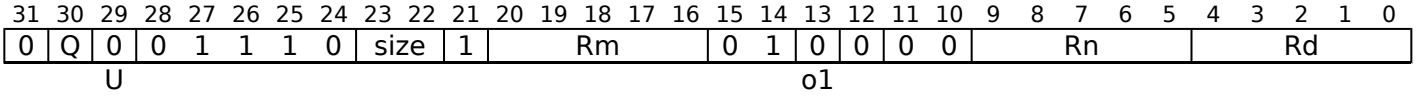
ADDHN, ADDHN2

Add returning High Narrow. This instruction adds each vector element in the first source SIMD&FP register to the corresponding vector element in the second source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

The results are truncated. For rounded results, see *RADDHN*.

The ADDHN instruction writes the vector to the lower half of the destination register and clears the upper half, while the ADDHN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



ADDHN{2} <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n, 2*datasize];
bits(2*datasize) operand2 = V[m, 2*datasize];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;

for e = 0 to elements-1
    element1 = Elem[operand1, e, 2*esize];
    element2 = Elem[operand2, e, 2*esize];
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    sum = sum + round_const;
    Elem[result, e, esize] = sum<2*esize-1:esize>;

Vpart[d, part, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

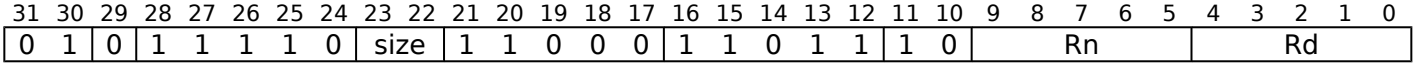
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ADDP (scalar)

Add Pair of elements (scalar). This instruction adds two vector elements in the source SIMD&FP register and writes the scalar result into the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



ADDP <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;

integer esize = 8 << UInt(size);
integer datasize = esize * 2;
```

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is the source arrangement specifier, encoded in “size”:

size	<T>
0x	RESERVED
10	RESERVED
11	2D

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
V[d, esize] = Reduce(ReduceOp_ADD, operand, esize);
```

Operational information

If PSTATE.DIT is 1:

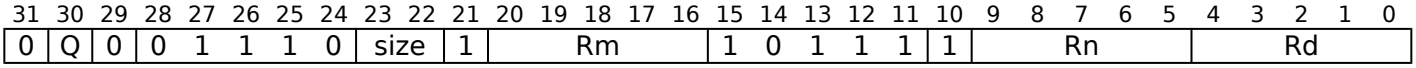
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ADDP (vector)

Add Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, adds each pair of values together, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



ADDP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[concat, 2*e, esize];
    element2 = Elem[concat, (2*e)+1, esize];
    Elem[result, e, esize] = element1 + element2;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.

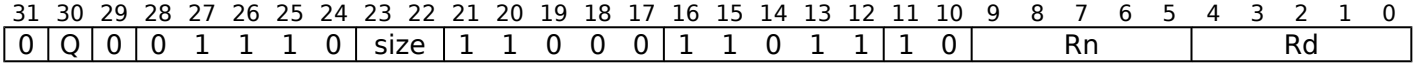
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ADDV

Add across Vector. This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



ADDV <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
V[d, esize] = Reduce(ReduceOp_ADD, operand, esize);
```

Operational information

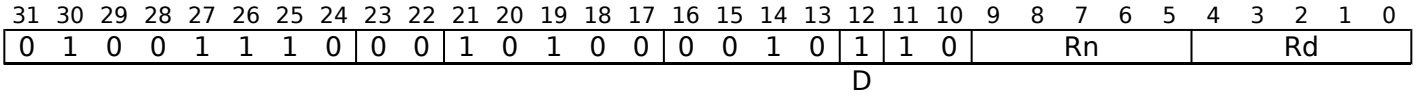
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

AESD

AES single round decryption.

Advanced SIMD
(FEAT_AES)



AESD <Vd>.16B, <Vn>.16B

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAESExt() then UNDEFINED;
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d, 128];
bits(128) operand2 = V[n, 128];
bits(128) result;
result = operand1 EOR operand2;
result = AESInvSubBytes(AESInvShiftRows(result));
V[d, 128] = result;
```

Operational information

If PSTATE.DIT is 1:

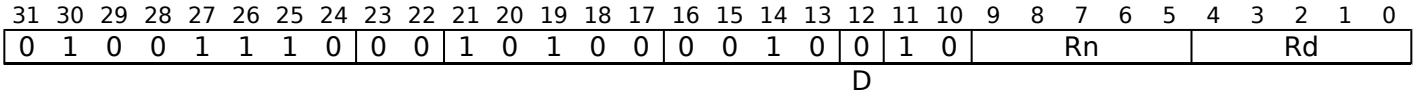
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

AESE

AES single round encryption.

Advanced SIMD
(FEAT_AES)



AESE <Vd>.16B, <Vn>.16B

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAESExt() then UNDEFINED;
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d, 128];
bits(128) operand2 = V[n, 128];
bits(128) result;
result = operand1 EOR operand2;
result = AESSubBytes(AESShiftRows(result));

V[d, 128] = result;
```

Operational information

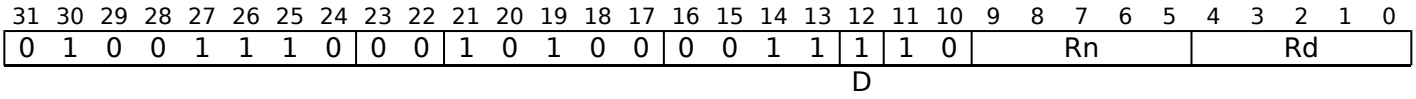
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

AESIMC

AES inverse mix columns.



AESIMC <Vd>.16B, <Vn>.16B

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAESExt() then UNDEFINED;
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn>	Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand = V[n, 128];
bits(128) result;
result = AESInvMixColumns(operand);
V[d, 128] = result;
```

Operational information

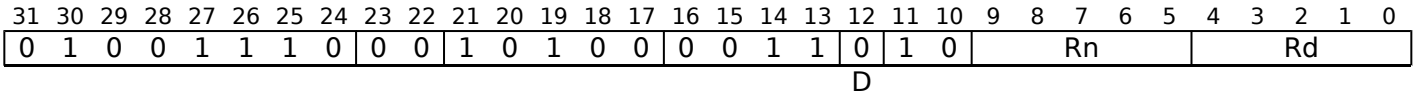
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

AESMC

AES mix columns.



AESMC <Vd>.16B, <Vn>.16B

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveAESExt() then UNDEFINED;
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn>	Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand = V[n, 128];
bits(128) result;
result = AESMixColumns(operand);
V[d, 128] = result;
```

Operational information

If PSTATE.DIT is 1:

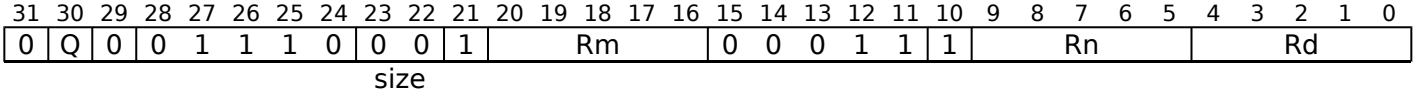
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

AND (vector)

Bitwise AND (vector). This instruction performs a bitwise AND between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



AND <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

result = operand1 AND operand2;
V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

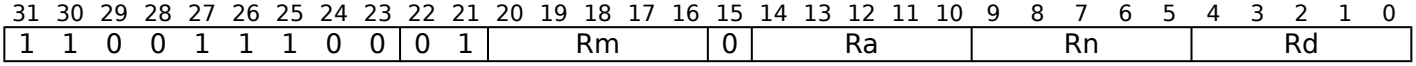
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BCAX

Bit Clear and Exclusive OR performs a bitwise AND of the 128-bit vector in a source SIMD&FP register and the complement of the vector in another source SIMD&FP register, then performs a bitwise exclusive OR of the resulting vector and the vector in a third source SIMD&FP register, and writes the result to the destination SIMD&FP register. This instruction is implemented only when *FEAT_SHA3* is implemented.

Advanced SIMD
(FEAT_SHA3)



BCAX <Vd>.16B, <Vn>.16B, <Vm>.16B, <Va>.16B

```
if !HaveSHA3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn>	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
<Va>	Is the name of the third SIMD&FP source register, encoded in the "Ra" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
bits(128) Va = V[a, 128];
V[d, 128] = Vn EOR (Vm AND NOT(Va));
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

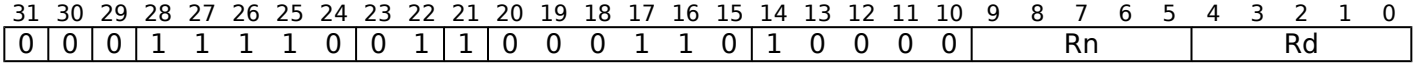
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BFCVT

Floating-point convert from single-precision to BFloat16 format (scalar) converts the single-precision floating-point value in the 32-bit SIMD&FP source register to BFloat16 format and writes the result in the 16-bit SIMD&FP destination register.

ID_AA64ISAR1_EL1.BF16 indicates whether this instruction is supported.

Single-precision to BFloat16
(FEAT_BF16)



BFCVT <Hd>, <Sn>

```
if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer d = UInt(Rd);
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEnabled64();

bits(32) operand = V[n, 32];
FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

Elem[result, 0, 16] = FPConvertBF(operand, fpcr);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

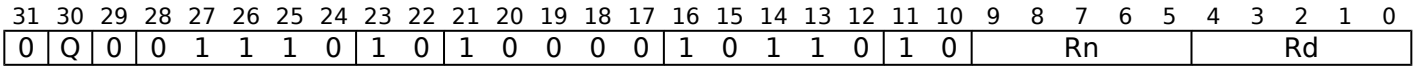
BFCVTN, BFCVTN2

Floating-point convert from single-precision to BFloat16 format (vector) reads each single-precision element in the SIMD&FP source vector, converts each value to BFloat16 format, and writes the results in the lower or upper half of the SIMD&FP destination vector. The result elements are half the width of the source elements.

The BFCVTN instruction writes the half-width results to the lower half of the destination vector and clears the upper half to zero, while the BFCVTN2 instruction writes the results to the upper half of the destination vector without affecting the other bits in the register.

Vector single-precision to BFloat16

(FEAT_BF16)



BFCVTN{2} <Vd>.<Ta>, <Vn>.4S

```
if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer d = UInt(Rd);
integer part = UInt(Q);
integer elements = 64 DIV 16;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	4H
1	8H

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(128) operand = V[n, 128];
bits(64) result;

for e = 0 to elements-1
    Elem[result, e, 16] = FPConvertBF(Elem[operand, e, 32], FPCR[]);

Vpart[d, part, 64] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BFDOT (by element)

BFloat16 floating-point dot product (vector, by element). This instruction delimits the source vectors into pairs of BFloat16 elements.

If FEAT_EBF16 is not implemented or *FPCR*.EBF is 0, irrespective of the other control bits in the *FPCR*, this instruction:

- Performs an unfused sum-of-products of each pair of adjacent BFloat16 elements in the first source vector with the specified pair of elements in the second source vector. The intermediate single-precision products are rounded before they are summed, and the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the first source vector.
- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- Does not modify the cumulative *FPSR* exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the *FPCR* trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
- Flushes denormalized inputs and results to zero, as if *FPCR*.{FZ, FIZ} is {1, 1}.
- Generates only the default NaN, as if *FPCR*.DN is 1.

If FEAT_EBF16 is implemented and *FPCR*.EBF is 1, then this instruction:

- Performs a fused sum-of-products of each pair of adjacent BFloat16 elements in the first source vector with the specified pair of elements in the second source vector. The intermediate single-precision products are not rounded before they are summed, but the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the first source vector.
- Generates only the default NaN, as if *FPCR*.DN is 1.
- Follows all other floating-point behaviors that apply to single-precision arithmetic, as controlled by the effective value of the *FPCR* in the current execution mode, and captured in the *FPSR*.

The BFloat16 pair within the second source vector is specified using an immediate index. The index range is from 0 to 3 inclusive. *ID_AA64ISAR1_EL1*.BF16 indicates whether this instruction is supported.

Vector
(FEAT_BF16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	0	1	L	M	Rm				1	1	1	1	H	0	Rn					Rd				

BFDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.2H[<index>]

```
if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer d = UInt(Rd);
integer i = UInt(H:L);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	4H
1	8H

<Vm> Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<index> Is the immediate index of a pair of 16-bit elements in the range 0 to 3, encoded in the "H:L" fields.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(128) operand2 = V[m, 128];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;

for e = 0 to elements-1
    bits(16) elt1_a = Elem[operand1, 2*e+0, 16];
    bits(16) elt1_b = Elem[operand1, 2*e+1, 16];
    bits(16) elt2_a = Elem[operand2, 2*i+0, 16];
    bits(16) elt2_b = Elem[operand2, 2*i+1, 16];

    bits(32) sum = Elem[operand3, e, 32];
    sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
    Elem[result, e, 32] = sum;

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BFDOT (vector)

BFloat16 floating-point dot product (vector). This instruction delimits the source vectors into pairs of BFloat16 elements.

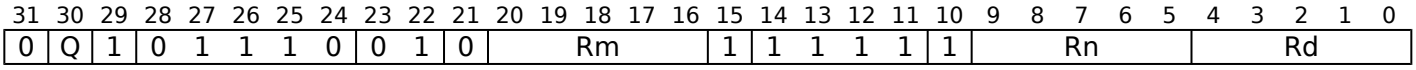
If FEAT_EBF16 is not implemented or *FPCR*.EBF is 0, irrespective of the other control bits in the *FPCR*, this instruction:

- Performs an unfused sum-of-products of each pair of adjacent BFloat16 elements in the source vectors. The intermediate single-precision products are rounded before they are summed, and the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the source vectors.
- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- Does not modify the cumulative *FPSR* exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the *FPCR* trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
- Flushes denormalized inputs and results to zero, as if *FPCR*.{FZ, FIZ} is {1, 1}.
- Generates only the default NaN, as if *FPCR*.DN is 1.

If FEAT_EBF16 is implemented and *FPCR*.EBF is 1, then this instruction:

- Performs a fused sum-of-products of each pair of adjacent BFloat16 elements in the source vectors. The intermediate single-precision products are not rounded before they are summed, but the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the source vectors.
- Generates only the default NaN, as if *FPCR*.DN is 1.
- Follows all other floating-point behaviors that apply to single-precision arithmetic, as controlled by the effective value of the *FPCR* in the current execution mode, and captured in the *FPSR*.

Vector
(FEAT_BF16)



BFDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	4H
1	8H

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;

for e = 0 to elements-1
    bits(16) elt1_a = Elem[operand1, 2*e+0, 16];
    bits(16) elt1_b = Elem[operand1, 2*e+1, 16];
    bits(16) elt2_a = Elem[operand2, 2*e+0, 16];
    bits(16) elt2_b = Elem[operand2, 2*e+1, 16];

    bits(32) sum = Elem[operand3, e, 32];
    sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
    Elem[result, e, 32] = sum;

V[d, datasize] = result;
```

BFMLALB, BFMLALT (by element)

BFloat16 floating-point widening multiply-add long (by element) widens the even-numbered (bottom) or odd-numbered (top) 16-bit elements in the first source vector, and the indexed element in the second source vector from Bfloat16 to single-precision format. The instruction then multiplies and adds these values without intermediate rounding to single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the first source vector.

ID_AA64ISAR1_EL1.BF16 indicates whether this instruction is supported.

Vector

(FEAT_BF16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	1	1	1	L	M	Rm				1	1	1	1	H	0	Rn					Rd				

BFMLAL<bt> <Vd>.4S, <Vn>.8H, <Vm>.H[<index>]

```
if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt('0':Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer elements = 128 DIV 32;
integer sel = UInt(Q);
```

Assembler Symbols

<bt> Is the bottom or top element specifier, encoded in “Q”:

Q	<bt>
0	B
1	T

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

<index> Is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(128) result;
bits(128) operand1 = V[n, 128];
bits(128) operand2 = V[m, 128];
bits(128) operand3 = V[d, 128];
bits(16) element2 = Elem[operand2, index, 16];

for e = 0 to elements-1
    bits(16) element1 = Elem[operand1, 2*e+sel, 16];
    bits(32) addend = Elem[operand3, e, 32];
    Elem[result, e, 32] = BFMulAddH(addend, element1, element2, FPCR[]);

V[d, 128] = result;
```

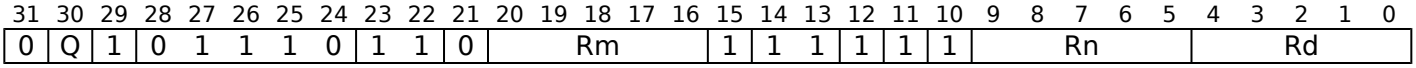
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BFMLALB, BFMLALT (vector)

BFloat16 floating-point widening multiply-add long (vector) widens the even-numbered (bottom) or odd-numbered (top) 16-bit elements in the first and second source vectors from Bfloat16 to single-precision format. The instruction then multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the source vectors.

ID_AA64ISAR1_EL1.BF16 indicates whether this instruction is supported.

Vector
(FEAT_BF16)



BFMLAL<bt> <Vd>.4S, <Vn>.8H, <Vm>.8H

```
if !HaveBF16Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer elements = 128 DIV 32;
integer sel = UInt(Q);
```

Assembler Symbols

<bt> Is the bottom or top element specifier, encoded in “Q”:

Q	<bt>
0	B
1	T

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(128) operand1 = V[n, 128];
bits(128) operand2 = V[m, 128];
bits(128) operand3 = V[d, 128];
bits(128) result;

for e = 0 to elements-1
    bits(16) element1 = Elem[operand1, 2*e+sel, 16];
    bits(16) element2 = Elem[operand2, 2*e+sel, 16];
    bits(32) addend = Elem[operand3, e, 32];
    Elem[result, e, 32] = BFMulAddH(addend, element1, element2, FPCR[]);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BFMMLA

BFloat16 floating-point matrix multiply-accumulate into 2x2 matrix.

If FEAT_EBF16 is not implemented or *FPCR*.EBF is 0, irrespective of the other control bits in the *FPCR*, this instruction:

- Performs two unfused sums-of-products within each two pairs of adjacent BFloat16 elements while multiplying the 2x4 matrix of BFloat16 values in the first source vector with the 4x2 matrix of BFloat16 values in the second source vector. The intermediate single-precision products are rounded before they are summed and the intermediate sum is rounded before accumulation into the 2x2 single-precision matrix in the destination vector. This is equivalent to accumulating two 2-way unfused dot products per destination element.
- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- Does not modify the cumulative *FPSR* exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the *FPCR* trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
- Flushes denormalized inputs and results to zero, as if *FPCR*.{FZ, FIZ} is {1, 1}.
- Generates only the default NaN, as if *FPCR*.DN is 1.

If FEAT_EBF16 is implemented and *FPCR*.EBF is 1, then this instruction:

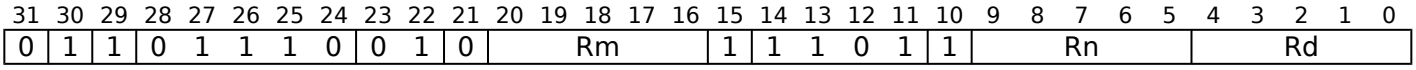
- Performs two fused sums-of-products within each two pairs of adjacent BFloat16 elements while multiplying the 2x4 matrix of BFloat16 values in the first source vector with the 4x2 matrix of BFloat16 values in the second source vector. The intermediate single-precision products are not rounded before they are summed, but the intermediate sum is rounded before accumulation into the 2x2 single-precision matrix in the destination vector. This is equivalent to accumulating two 2-way fused dot products per destination element.
- Generates only the default NaN, as if *FPCR*.DN is 1.
- Follows all other floating-point behaviors that apply to single-precision arithmetic, as controlled by the effective value of the *FPCR* in the current execution mode, and captured in the *FPSR*.

Note

Arm expects that the BFMMLA instruction will deliver a peak BFloat16 multiply throughput that is at least as high as can be achieved using two BFDOT instructions, with a goal that it should have significantly higher throughput.

Vector

(FEAT_BF16)



BFMMLA <Vd>.4S, <Vn>.8H, <Vm>.8H

```
if !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

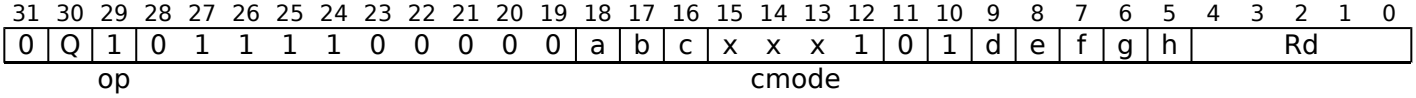
Operation

```
CheckFPAdvSIMDEnabled64();
bits(128) op1 = V[n, 128];
bits(128) op2 = V[m, 128];
bits(128) acc = V[d, 128];

V[d, 128] = BFMatMulAdd(acc, op1, op2);
```

BIC (vector, immediate)

Bitwise bit Clear (vector, immediate). This instruction reads each vector element from the destination SIMD&FP register, performs a bitwise AND between each result and the complement of an immediate constant, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



16-bit (cmode == 10x1)

```
BIC <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

32-bit (cmode == 0xx1)

```
BIC <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
    when '0xx01' operation = ImmediateOp_MVNI;
    when '0xx11' operation = ImmediateOp_BIC;
    when '10x01' operation = ImmediateOp_MVNI;
    when '10x11' operation = ImmediateOp_BIC;
    when '110x1' operation = ImmediateOp_MVNI;
    when '1110x' operation = ImmediateOp_MOVI;
    when '11111'
        // FMOV Dn,#imm is in main FP instruction set
        if Q == '0' then UNDEFINED;
        operation = ImmediateOp_MOVI;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP register, encoded in the "Rd" field.

<T> For the 16-bit variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the 32-bit variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	2S
1	4S

<imm8> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

<amount> For the 16-bit variant: is the shift amount encoded in “cmode<1>”:

cmode<1>	<amount>
0	0
1	8

defaulting to 0 if LSL is omitted.

For the 32-bit variant: is the shift amount encoded in “cmode<2:1>”:

cmode<2:1>	<amount>
00	0
01	8
10	16
11	24

defaulting to 0 if LSL is omitted.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
    when ImmediateOp_MOVI
        result = imm;
    when ImmediateOp_MVNI
        result = NOT(imm);
    when ImmediateOp_ORR
        operand = V[rd, datasize];
        result = operand OR imm;
    when ImmediateOp_BIC
        operand = V[rd, datasize];
        result = operand AND NOT(imm);

V[rd, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

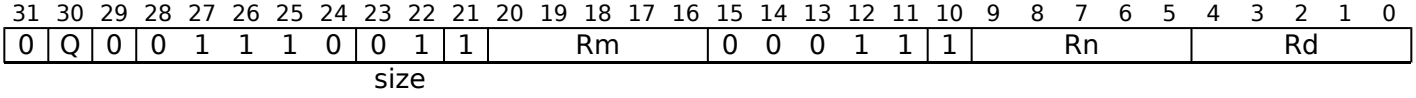
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BIC (vector, register)

Bitwise bit Clear (vector, register). This instruction performs a bitwise AND between the first source SIMD&FP register and the complement of the second source SIMD&FP register, and writes the result to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



BIC <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

operand2 = NOT(operand2);

result = operand1 AND operand2;
V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

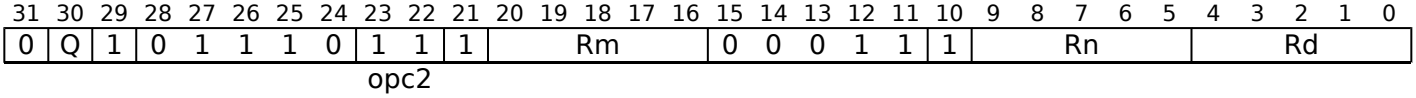
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BIF

Bitwise Insert if False. This instruction inserts each bit from the first source SIMD&FP register into the destination SIMD&FP register if the corresponding bit of the second source SIMD&FP register is 0, otherwise leaves the bit in the destination register unchanged.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



BIF <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand3;
bits(datasize) operand4 = V[n, datasize];

operand1 = V[d, datasize];
operand3 = NOT(V[m, datasize]);

V[d, datasize] = operand1 EOR ((operand1 EOR operand4) AND operand3);
```

Operational information

If PSTATE.DIT is 1:

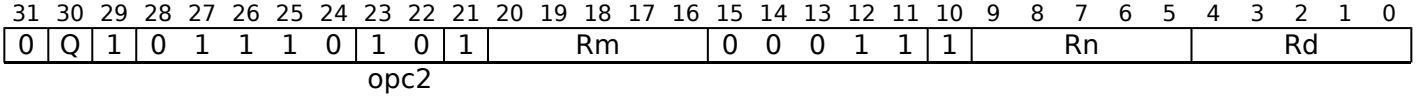
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BIT

Bitwise Insert if True. This instruction inserts each bit from the first source SIMD&FP register into the SIMD&FP destination register if the corresponding bit of the second source SIMD&FP register is 1, otherwise leaves the bit in the destination register unchanged.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



BIT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand3;
bits(datasize) operand4 = V[n, datasize];

operand1 = V[d, datasize];
operand3 = V[m, datasize];
V[d, datasize] = operand1 EOR ((operand1 EOR operand4) AND operand3);
```

Operational information

If PSTATE.DIT is 1:

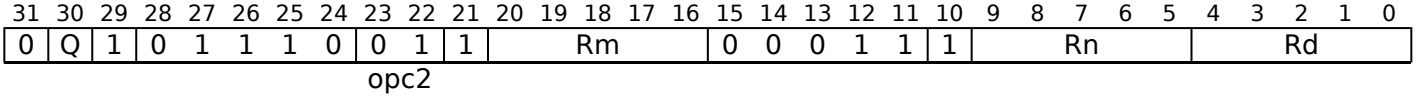
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BSL

Bitwise Select. This instruction sets each bit in the destination SIMD&FP register to the corresponding bit from the first source SIMD&FP register when the original destination bit was 1, otherwise from the second source SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



BSL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand3;
bits(datasize) operand4 = V[n, datasize];

operand1 = V[m, datasize];
operand3 = V[d, datasize];
V[d, datasize] = operand1 EOR ((operand1 EOR operand4) AND operand3);
```

Operational information

If PSTATE.DIT is 1:

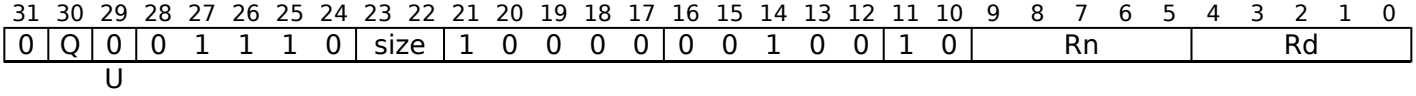
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CLS (vector)

Count Leading Sign bits (vector). This instruction counts the number of consecutive bits following the most significant bit that are the same as the most significant bit in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The count does not include the most significant bit itself.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



CLS <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CountOp countop = if U == '1' then CountOp_CLZ else CountOp_CLS;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;

integer count;
for e = 0 to elements-1
    if countop == CountOp_CLS then
        count = CountLeadingSignBits(Elem[operand, e, esize]);
    else
        count = CountLeadingZeroBits(Elem[operand, e, esize]);
    Elem[result, e, esize] = count<esize-1:0>;
V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

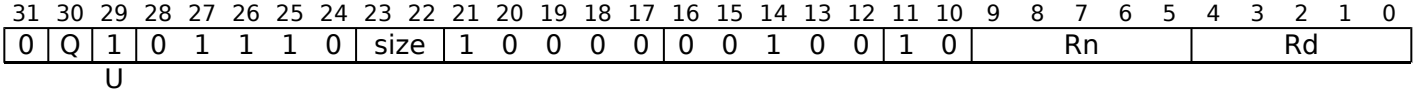
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

CLZ (vector)

Count Leading Zero bits (vector). This instruction counts the number of consecutive zeros, starting from the most significant bit, in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



CLZ <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CountOp countop = if U == '1' then CountOp_CLZ else CountOp_CLS;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;

integer count;
for e = 0 to elements-1
    if countop == CountOp_CLS then
        count = CountLeadingSignBits(Elem[operand, e, esize]);
    else
        count = CountLeadingZeroBits(Elem[operand, e, esize]);
    Elem[result, e, esize] = count<esize-1:0>;
V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

- The values of the NZCV flags.

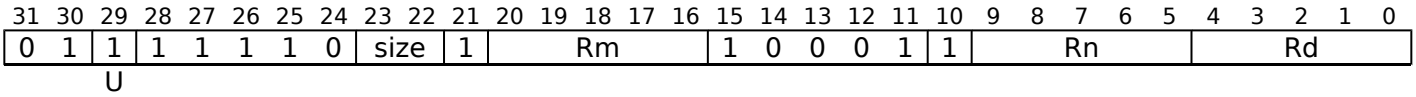
CMEQ (register)

Compare bitwise Equal (vector). This instruction compares each vector element from the first source SIMD&FP register with the corresponding vector element from the second source SIMD&FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

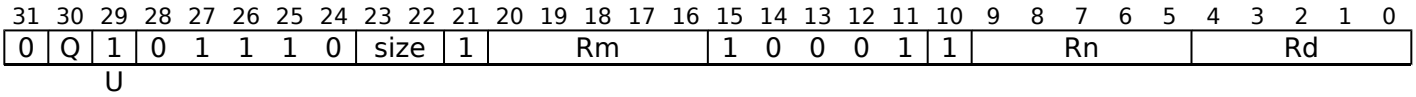
Scalar



CMEQ <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean and_test = (U == '0');
```

Vector



CMEQ <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean and_test = (U == '0');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if and_test then
        test_passed = !IsZero(element1 AND element2);
    else
        test_passed = (element1 == element2);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

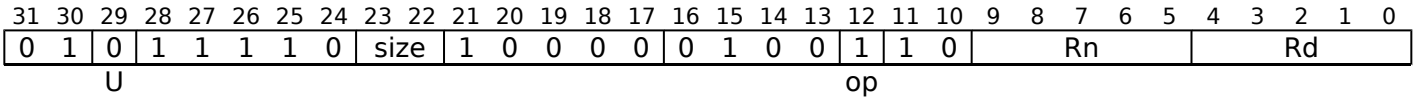
CMEQ (zero)

Compare bitwise Equal to zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

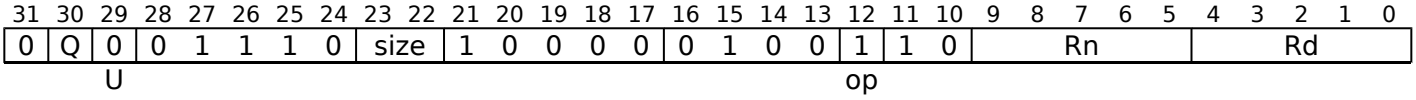
Scalar



CMEQ <V><d>, <V><n>, #0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector



CMEQ <Vd>.<T>, <Vn>.<T>, #0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
boolean test_passed;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

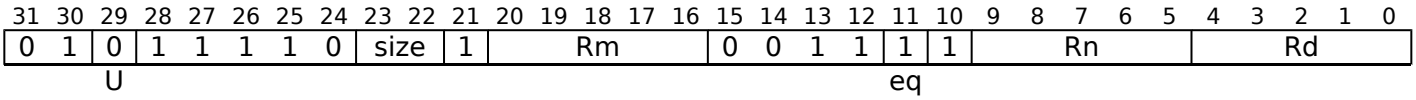
CMGE (register)

Compare signed Greater than or Equal (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first signed integer value is greater than or equal to the second signed integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

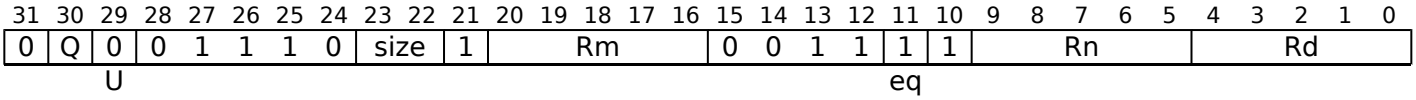
Scalar



CMGE <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

Vector



CMGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

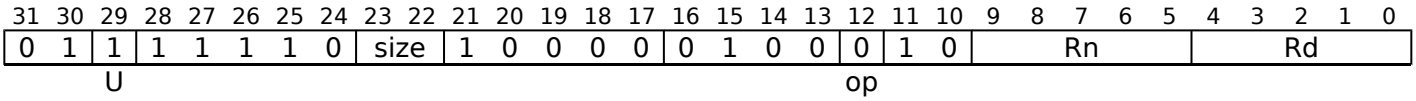
CMGE (zero)

Compare signed Greater than or Equal to zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is greater than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

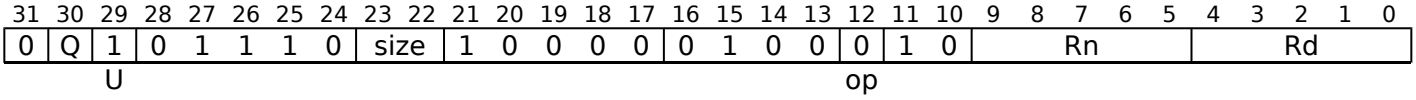
Scalar



CMGE <V><d>, <V><n>, #0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector



CMGE <Vd>.<T>, <Vn>.<T>, #0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
boolean test_passed;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

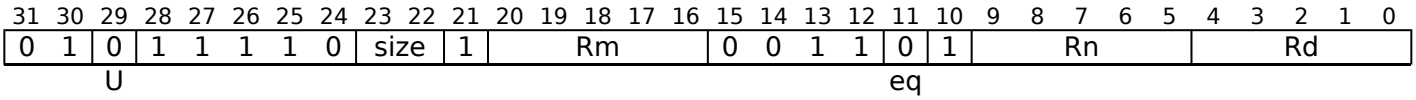
CMGT (register)

Compare signed Greater than (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first signed integer value is greater than the second signed integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

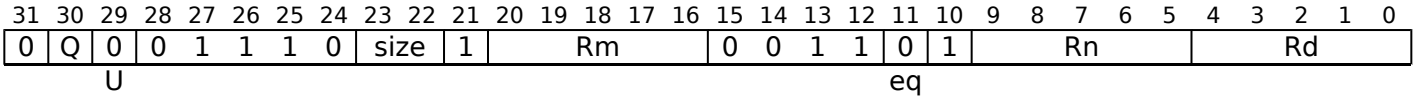
Scalar



CMGT <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

Vector



CMGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

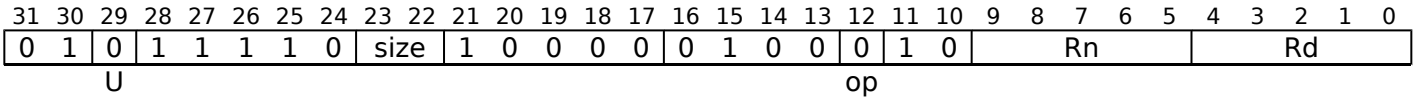
CMGT (zero)

Compare signed Greater than zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is greater than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

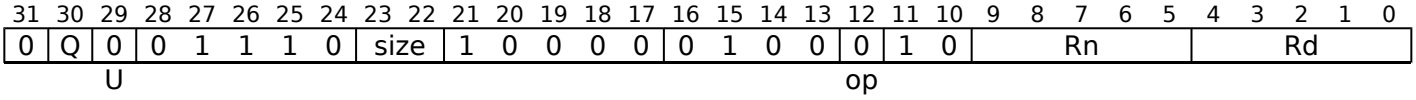
Scalar



CMGT <V><d>, <V><n>, #0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector



CMGT <Vd>.<T>, <Vn>.<T>, #0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
boolean test_passed;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

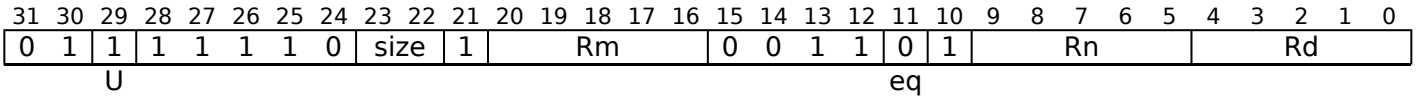
CMHI (register)

Compare unsigned Higher (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first unsigned integer value is greater than the second unsigned integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

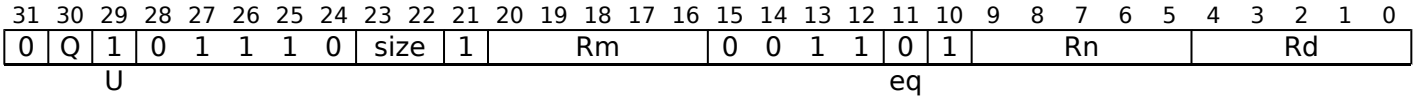
Scalar



CMHI <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

Vector



CMHI <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

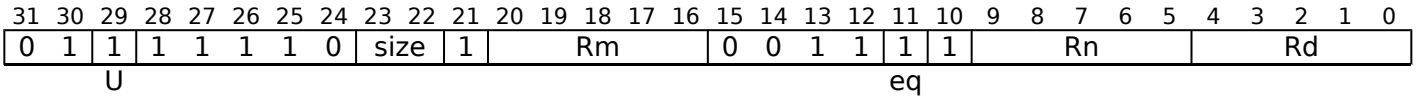
CMHS (register)

Compare unsigned Higher or Same (vector). This instruction compares each vector element in the first source SIMD&FP register with the corresponding vector element in the second source SIMD&FP register and if the first unsigned integer value is greater than or equal to the second unsigned integer value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

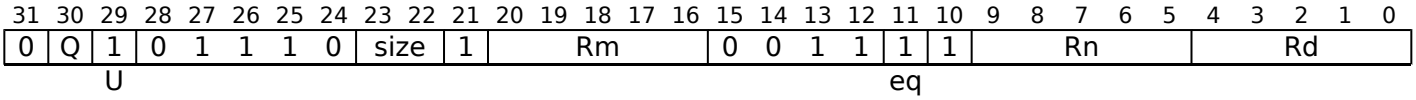
Scalar



CMHS <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

Vector



CMHS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean cmp_eq = (eq == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    test_passed = if cmp_eq then element1 >= element2 else element1 > element2;
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

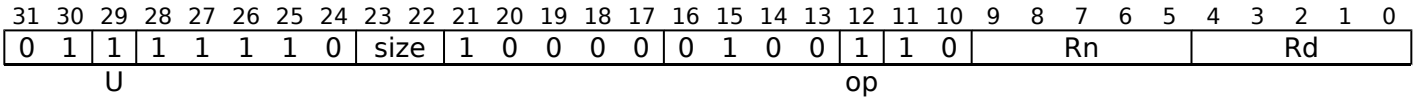
CMLE (zero)

Compare signed Less than or Equal to zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is less than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

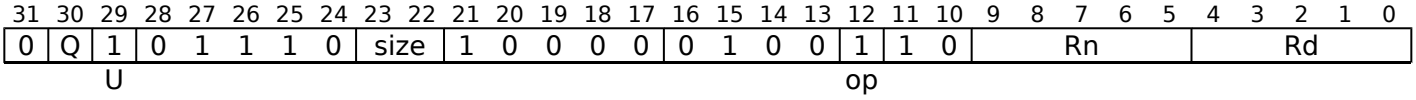
Scalar



CMLE <V><d>, <V><n>, #0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector



CMLE <Vd>.<T>, <Vn>.<T>, #0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
boolean test_passed;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

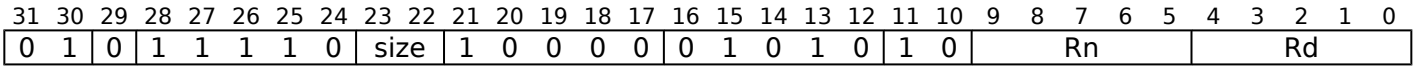
CMLT (zero)

Compare signed Less than zero (vector). This instruction reads each vector element in the source SIMD&FP register and if the signed integer value is less than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

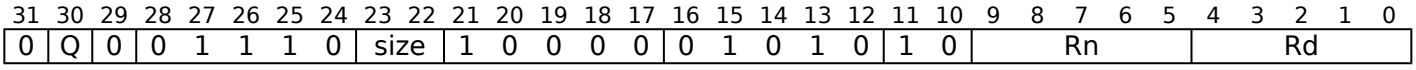
Scalar



CMLT <V><d>, <V><n>, #0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

CompareOp comparison = CompareOp_LT;
```

Vector



CMLT <Vd>.<T>, <Vn>.<T>, #0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison = CompareOp_LT;
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
boolean test_passed;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    case comparison of
        when CompareOp_GT test_passed = element > 0;
        when CompareOp_GE test_passed = element >= 0;
        when CompareOp_EQ test_passed = element == 0;
        when CompareOp_LE test_passed = element <= 0;
        when CompareOp_LT test_passed = element < 0;
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

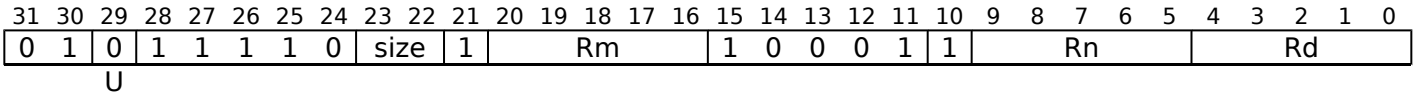
CMTST

Compare bitwise Test bits nonzero (vector). This instruction reads each vector element in the first source SIMD&FP register, performs an AND with the corresponding vector element in the second source SIMD&FP register, and if the result is not zero, sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

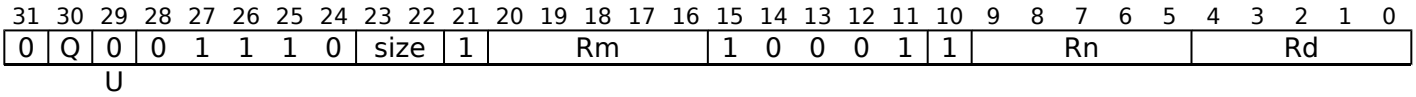
Scalar



CMTST <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean and_test = (U == '0');
```

Vector



CMTST <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean and_test = (U == '0');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
boolean test_passed;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if and_test then
        test_passed = !IsZero(element1 AND element2);
    else
        test_passed = (element1 == element2);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

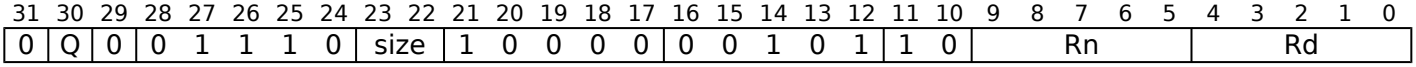
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CNT

Population Count per byte. This instruction counts the number of bits that have a value of one in each vector element in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



CNT <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '00' then UNDEFINED;
integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	x	RESERVED
1x	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;

integer count;
for e = 0 to elements-1
    count = BitCount(Elem[operand, e, esize]);
    Elem[result, e, esize] = count<esize-1:0>;
V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

DUP (element)

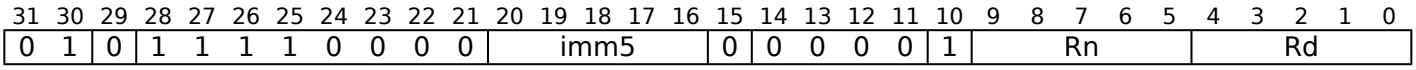
Duplicate vector element to vector or scalar. This instruction duplicates the vector element at the specified element index in the source SIMD&FP register into a scalar or each element in a vector, and writes the result to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (scalar).

It has encodings from 2 classes: Scalar and Vector

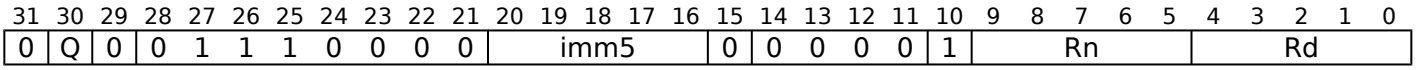
Scalar



DUP <V><d>, <Vn>.<T>[<index>]

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;

integer index = UInt(imm5<4:size+1>);
integer idxdsize = if imm5<4> == '1' then 128 else 64;

integer esize = 8 << size;
integer datasize = esize;
integer elements = 1;
```

Vector



DUP <Vd>.<T>, <Vn>.<Ts>[<index>]

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;

integer index = UInt(imm5<4:size+1>);
integer idxdsize = if imm5<4> == '1' then 128 else 64;

if size == 3 && Q == '0' then UNDEFINED;
integer esize = 8 << size;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<T> For the scalar variant: is the element width specifier, encoded in “imm5”:

imm5	<T>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

For the vector variant: is an arrangement specifier, encoded in “imm5:Q”:

imm5	Q	<T>
x0000	x	RESERVED
xxxx1	0	8B
xxxx1	1	16B
xxx10	0	4H
xxx10	1	8H
xx100	0	2S
xx100	1	4S
x1000	0	RESERVED
x1000	1	2D

<Ts> Is an element size specifier, encoded in “imm5”:

imm5	<Ts>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<V> Is the destination width specifier, encoded in “imm5”:

imm5	<V>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<index> Is the element index encoded in “imm5”:

imm5	<index>
x0000	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>
xx100	imm5<4:3>
x1000	imm5<4>

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(idxdsize) operand = V[n, idxdsize];
bits(datasize) result;
bits(esize) element;

element = Elem[operand, index, esize];
for e = 0 to elements-1
    Elem[result, e, esize] = element;
V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

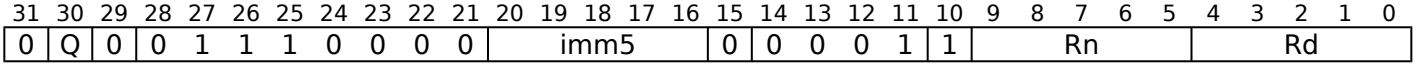
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

- The values of the NZCV flags.

DUP (general)

Duplicate general-purpose register to vector. This instruction duplicates the contents of the source general-purpose register into a scalar or each element in a vector, and writes the result to the SIMD&FP destination register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



DUP <Vd>.<T>, <R><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;

// imm5<4:size+1> is IGNORED

if size == 3 && Q == '0' then UNDEFINED;
integer esize = 8 << size;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “imm5:Q”:

imm5	Q	<T>
x0000	x	RESERVED
xxxx1	0	8B
xxxx1	1	16B
xxx10	0	4H
xxx10	1	8H
xx100	0	2S
xx100	1	4S
x1000	0	RESERVED
x1000	1	2D

<R> Is the width specifier for the general-purpose source register, encoded in “imm5”:

imm5	<R>
x0000	RESERVED
xxxx1	W
xxx10	W
xx100	W
x1000	X

Unspecified bits in "imm5" are ignored but should be set to zero by an assembler.

<n> Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(esize) element = X[n, esize];
bits(datasize) result;

for e = 0 to elements-1
    Elem[result, e, esize] = element;
V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

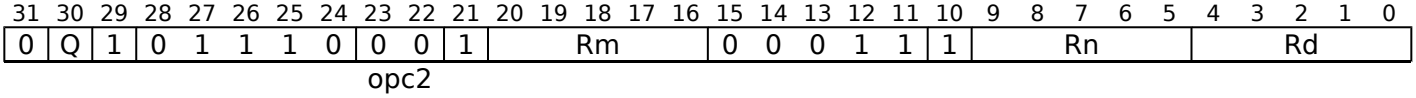
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

EOR (vector)

Bitwise Exclusive OR (vector). This instruction performs a bitwise Exclusive OR operation between the two source SIMD&FP registers, and places the result in the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



EOR <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1;
bits(datasize) operand2;
bits(datasize) operand3;
bits(datasize) operand4 = V[n, datasize];

operand1 = V[m, datasize];
operand2 = Zeros(datasize);
operand3 = Ones(datasize);
V[d, datasize] = operand1 EOR ((operand2 EOR operand4) AND operand3);
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

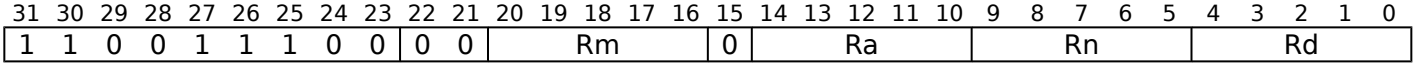
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

EOR3

Three-way Exclusive OR performs a three-way exclusive OR of the values in the three source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction is implemented only when *FEAT_SHA3* is implemented.

Advanced SIMD
(FEAT_SHA3)



EOR3 <Vd>.16B, <Vn>.16B, <Vm>.16B, <Va>.16B

```
if !HaveSHA3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn>	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
<Va>	Is the name of the third SIMD&FP source register, encoded in the "Ra" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
bits(128) Va = V[a, 128];
V[d, 128] = Vn EOR Vm EOR Va;
```

Operational information

If PSTATE.DIT is 1:

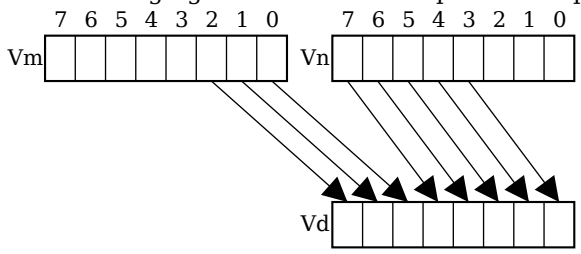
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

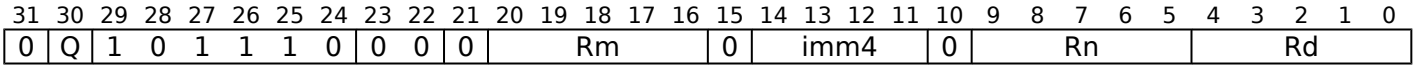
EXT

Extract vector from pair of vectors. This instruction extracts the lowest vector elements from the second source SIMD&FP register and the highest vector elements from the first source SIMD&FP register, concatenates the results into a vector, and writes the vector to the destination SIMD&FP register vector. The index value specifies the lowest vector element to extract from the first source register, and consecutive elements are extracted from the first, then second, source registers until the destination vector is filled.

The following figure shows an example of the operation of EXT doubleword operation for Q = 0 and imm4<2:0> = 3.



Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



EXT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<index>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if Q == '0' && imm4<3> == '1' then UNDEFINED;

integer datasize = if Q == '1' then 128 else 64;
integer position = UInt(imm4) << 3;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<index> Is the lowest numbered byte element to be extracted, encoded in “Q:imm4”:

Q	imm4<3>	<index>
0	0	imm4<2:0>
0	1	RESERVED
1	x	imm4

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) hi = V[m, datasize];
bits(datasize) lo = V[n, datasize];
bits(datasize*2) concat = hi:lo;

V[d, datasize] = concat<position+datasize-1:position>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FABD

Floating-point Absolute Difference (vector). This instruction subtracts the floating-point values in the elements of the second source SIMD&FP register, from the corresponding floating-point values in the elements of the first source SIMD&FP register, places the absolute value of each result in a vector, and writes the vector to the destination SIMD&FP register.

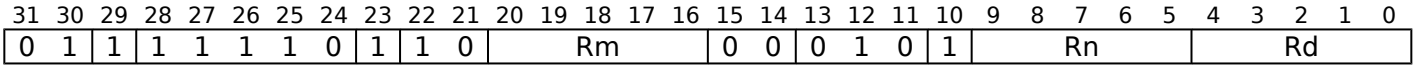
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

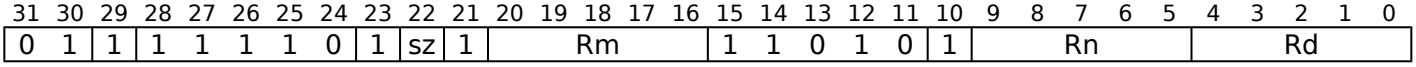
(FEAT_FP16)



FABD <Hd>, <Hn>, <Hm>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean abs = TRUE;
```

Scalar single-precision and double-precision

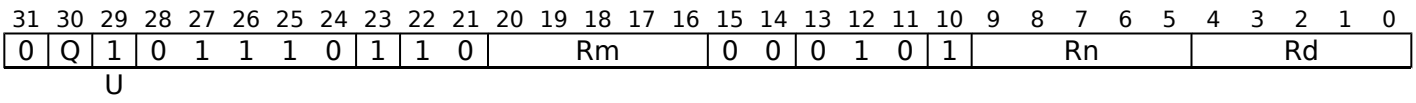


FABD <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean abs = TRUE;
```

Vector half precision

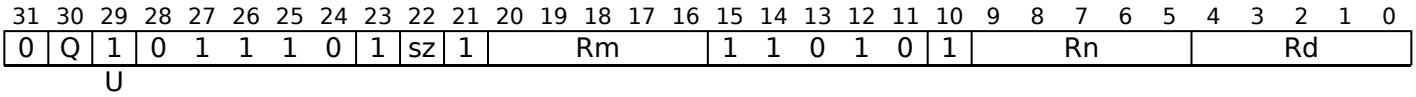
(FEAT_FP16)



FABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');
```

Vector single-precision and double-precision



FABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

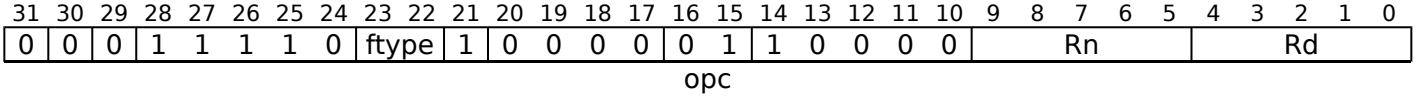
```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
bits(esize) diff;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    diff = FPSub(element1, element2, fpcr);
    Elem[result, e, esize] = if abs then FPAbs(diff) else diff;

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FABS (scalar)

Floating-point Absolute value (scalar). This instruction calculates the absolute value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)
(FEAT_FP16)

```
FABS <Hd>, <Hn>
```

Single-precision (ftype == 00)

```
FABS <Sd>, <Sn>
```

Double-precision (ftype == 01)

```
FABS <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

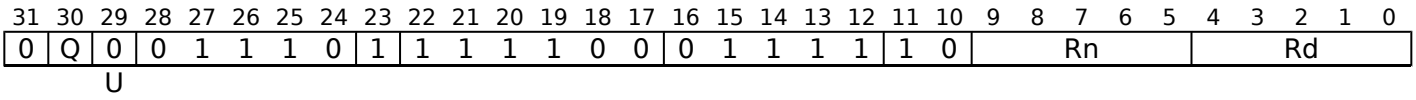
Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else 0<127:0>;

bits(esize) operand = V[n, esize];

Elem[result, 0, esize] = FPAbs(operand);
V[d, 128] = result;
```

FABS (vector)

Floating-point Absolute value (vector). This instruction calculates the absolute value of each vector element in the source SIMD&FP register, writes the result to a vector, and writes the vector to the destination SIMD&FP register.
Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.
It has encodings from 2 classes: Half-precision and Single-precision and double-precision

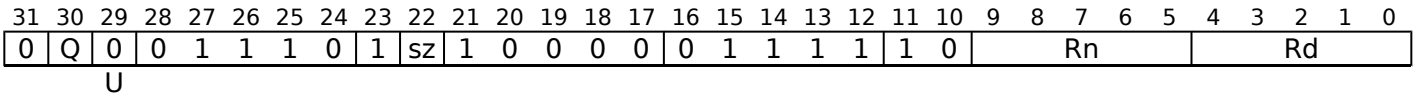
Half-precision
(FEAT_FP16)



FABS <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Single-precision and double-precision



FABS <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    if neg then
        element = FPNeg(element);
    else
        element = FPAbs(element);
    Elem[result, e, esize] = element;

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FACGE

Floating-point Absolute Compare Greater than or Equal (vector). This instruction compares the absolute value of each floating-point value in the first source SIMD&FP register with the absolute value of the corresponding floating-point value in the second source SIMD&FP register and if the first value is greater than or equal to the second value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

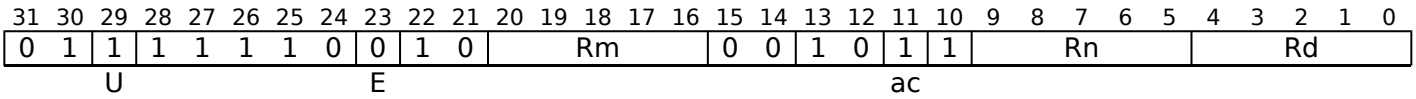
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

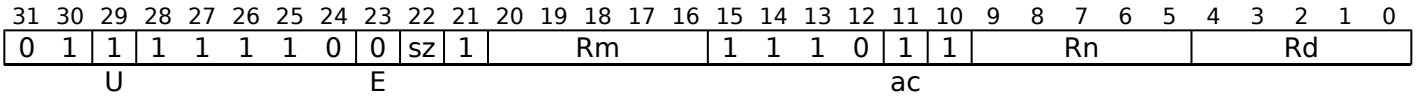
(FEAT_FP16)



FACGE <Hd>, <Hn>, <Hm>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

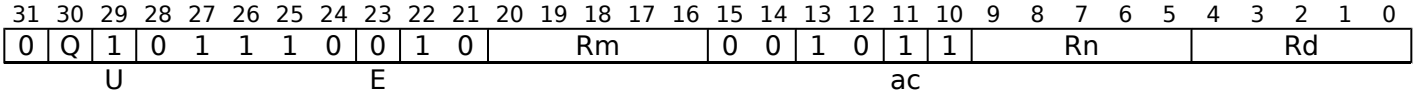
Scalar single-precision and double-precision



FACGE <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

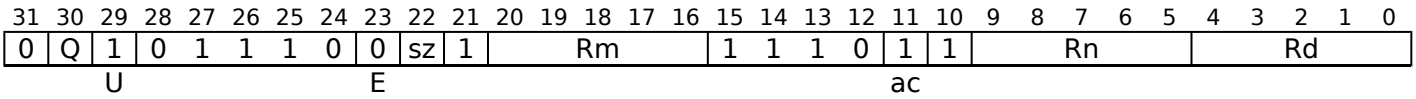
Vector half precision
(FEAT_FP16)



FACGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Vector single-precision and double-precision



FACGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
boolean test_passed;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[m, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, fpcr);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, fpcr);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, fpcr);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FACGT

Floating-point Absolute Compare Greater than (vector). This instruction compares the absolute value of each vector element in the first source SIMD&FP register with the absolute value of the corresponding vector element in the second source SIMD&FP register and if the first value is greater than the second value sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

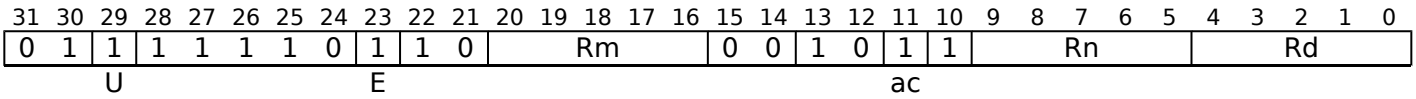
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

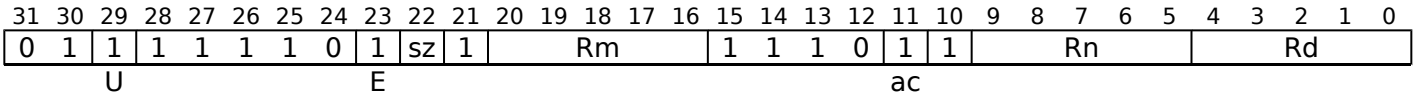
(FEAT_FP16)



FACGT <Hd>, <Hn>, <Hm>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

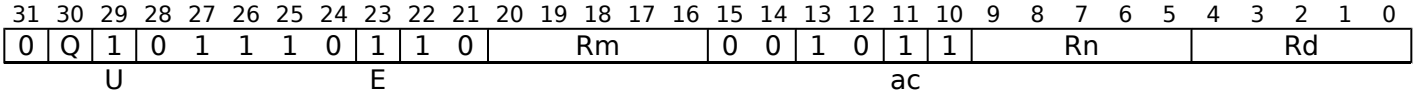
Scalar single-precision and double-precision



FACGT <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

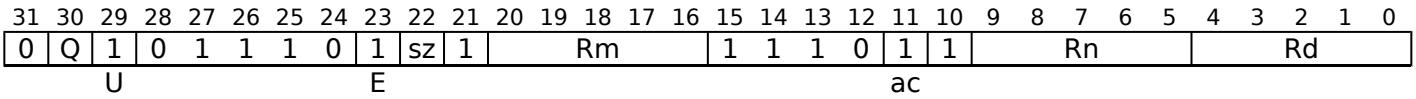
Vector half precision
(FEAT_FP16)



FACGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Vector single-precision and double-precision



FACGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
boolean test_passed;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[m, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, fpcr);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, fpcr);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, fpcr);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, 128] = result;
```

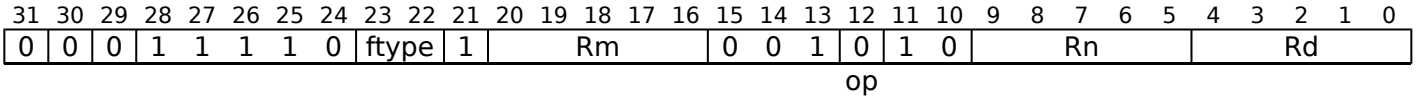
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FADD (scalar)

Floating-point Add (scalar). This instruction adds the floating-point values of the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)

(FEAT_FP16)

```
FADD <Hd>, <Hn>, <Hm>
```

Single-precision (ftype == 00)

```
FADD <Sd>, <Sn>, <Sm>
```

Double-precision (ftype == 01)

```
FADD <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPEnabled64();
bits(esize) operand1 = V[n, esize];
bits(esize) operand2 = V[m, esize];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

Elem[result, 0, esize] = FPAdd(operand1, operand2, fpcr);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FADD (vector)

Floating-point Add (vector). This instruction adds corresponding vector elements in the two source SIMD&FP registers, writes the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

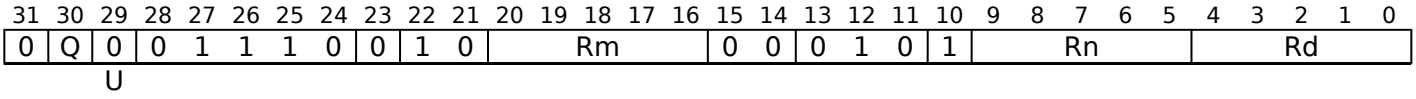
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

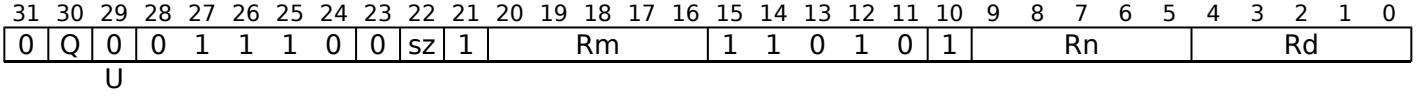
(FEAT_FP16)



FADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
```

Single-precision and double-precision



FADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPAdd(element1, element2, FPCR[]);

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FADDP (scalar)

Floating-point Add Pair of elements (scalar). This instruction adds two floating-point vector elements in the source SIMD&FP register and writes the scalar result into the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	0	1	1	0	Rn					Rd				

FADDP <V><d>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer esize = 16;
if sz == '1' then UNDEFINED;
integer datasize = 32;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	0	1	1	0	Rn					Rd				

FADDP <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
```

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	H
1	RESERVED

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2H
1	RESERVED

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in "sz":

sz	<T>
0	2S
1	2D

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
V[d, esize] = Reduce(ReduceOp_FADD, operand, esize);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FADDP (vector)

Floating-point Add Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, adds each pair of values together, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

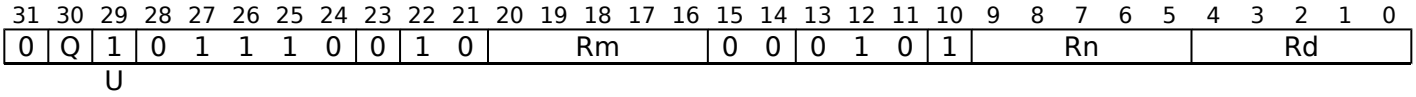
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

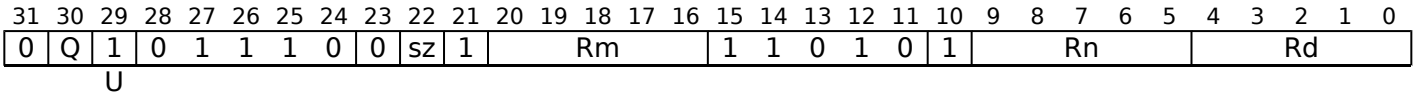
(FEAT_FP16)



FADDP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
```

Single-precision and double-precision



FADDP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPAdd(element1, element2, FPCR[]);

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCADD

Floating-point Complex Add.

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on the corresponding complex number element pairs from the two source registers:

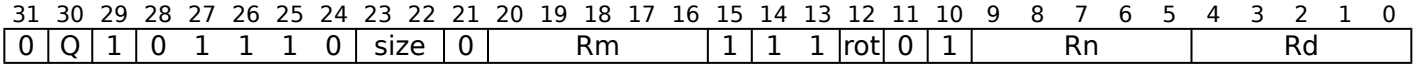
- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 90 or 270 degrees.
- The rotated complex number is added to the complex number from the first source register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_FCMA)



FCADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<rotate>

```
if !HaveFCADDExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '00' then UNDEFINED;
if Q == '0' && size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<rotate> Is the rotation, encoded in “rot”:

rot	<rotate>
0	90
1	270

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element3;

for e = 0 to (elements DIV 2)-1
    case rot of
        when '0'
            element1 = FPNeg(Elem[operand2, e*2+1, esize]);
            element3 = Elem[operand2, e*2, esize];
        when '1'
            element1 = Elem[operand2, e*2+1, esize];
            element3 = FPNeg(Elem[operand2, e*2, esize]);
    Elem[result, e*2, esize] = FPAdd(Elem[operand1, e*2, esize], element1, FPCR[]);
    Elem[result, e*2+1, esize] = FPAdd(Elem[operand1, e*2+1, esize], element3, FPCR[]);

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

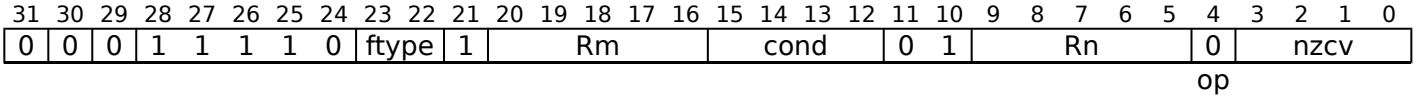
FCCMP

Floating-point Conditional quiet Compare (scalar). This instruction compares the two SIMD&FP source register values and writes the result to the *PSTATE*.{N, Z, C, V} flags. If the condition does not pass then the *PSTATE*.{N, Z, C, V} flags are set to the flag bit specifier.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is a signaling NaN.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)

(FEAT_FP16)

```
FCCMP <Hn>, <Hm>, #<nzcv>, <cond>
```

Single-precision (ftype == 00)

```
FCCMP <Sn>, <Sm>, #<nzcv>, <cond>
```

Double-precision (ftype == 01)

```
FCCMP <Dn>, <Dm>, #<nzcv>, <cond>
```

```
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;

bits(4) flags = nzcv;
```

Assembler Symbols

<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<nzcv>	Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
<cond>	Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

```
CheckFPEnabled64();

bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2;

operand2 = V[m, datasize];

if ConditionHolds(cond) then
    flags = FPCompare(operand1, operand2, FALSE, FPCR[]);
PSTATE.<N,Z,C,V> = flags;
```

Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the *PSTATE* condition flags to N=0, Z=0, C=1, and V=1.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

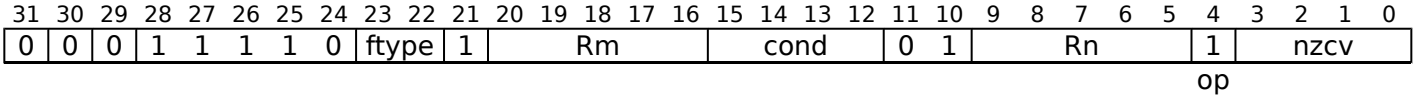
FCCMPE

Floating-point Conditional signaling Compare (scalar). This instruction compares the two SIMD&FP source register values and writes the result to the *PSTATE*.{N, Z, C, V} flags. If the condition does not pass then the *PSTATE*.{N, Z, C, V} flags are set to the flag bit specifier.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is any type of NaN.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)

(FEAT_FP16)

```
FCCMPE <Hn>, <Hm>, #<nzcv>, <cond>
```

Single-precision (ftype == 00)

```
FCCMPE <Sn>, <Sm>, #<nzcv>, <cond>
```

Double-precision (ftype == 01)

```
FCCMPE <Dn>, <Dm>, #<nzcv>, <cond>
```

```
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;

bits(4) flags = nzcv;
```

Assembler Symbols

<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<nzcv>	Is the flag bit specifier, an immediate in the range 0 to 15, giving the alternative state for the 4-bit NZCV condition flags, encoded in the "nzcv" field.
<cond>	Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

```
CheckFPEnabled64();

bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2;

operand2 = V[m, datasize];

if ConditionHolds(cond) then
    flags = FPCompare(operand1, operand2, TRUE, FPCR[]);
PSTATE.<N,Z,C,V> = flags;
```

Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the *PSTATE* condition flags to N=0, Z=0, C=1, and V=1.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCMEQ (register)

Floating-point Compare Equal (vector). This instruction compares each floating-point value from the first source SIMD&FP register, with the corresponding floating-point value from the second source SIMD&FP register, and if the comparison is equal sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

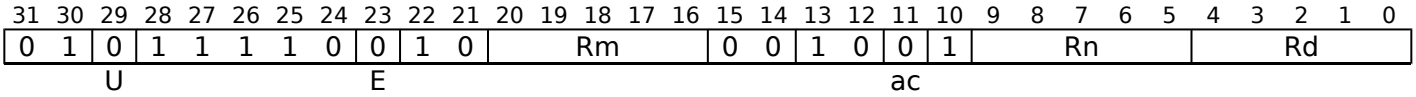
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

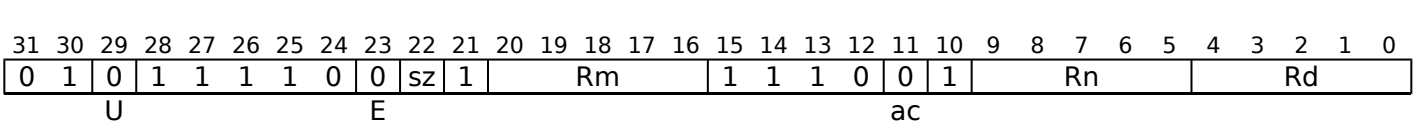
(FEAT_FP16)



FCMEQ <Hd>, <Hn>, <Hm>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

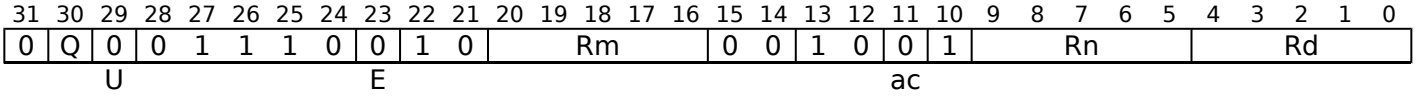
Scalar single-precision and double-precision



FCMEQ <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

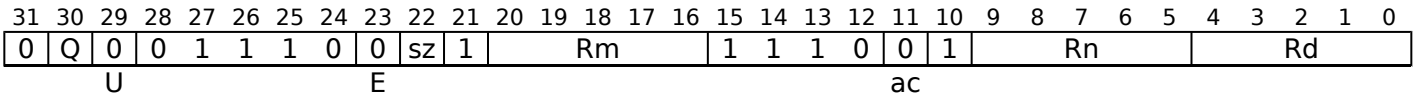
Vector half precision
(FEAT_FP16)



FCMEQ <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Vector single-precision and double-precision



FCMEQ <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
boolean test_passed;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[m, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, fpcr);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, fpcr);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, fpcr);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCMEQ (zero)

Floating-point Compare Equal to zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

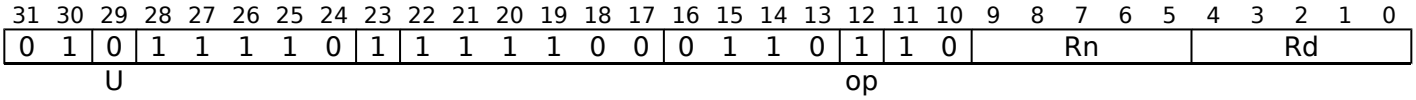
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

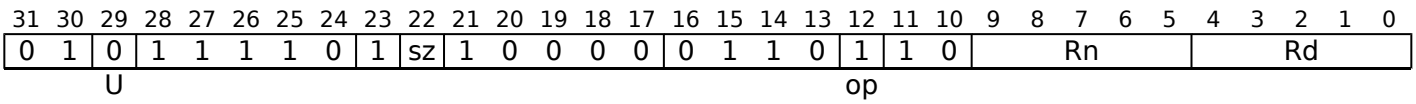
(FEAT_FP16)



FCMEQ <Hd>, <Hn>, #0.0

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

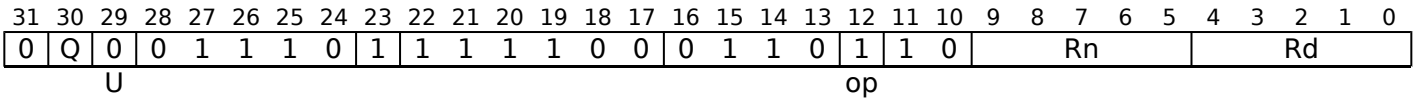
Scalar single-precision and double-precision



FCMEQ <V><d>, <V><n>, #0.0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

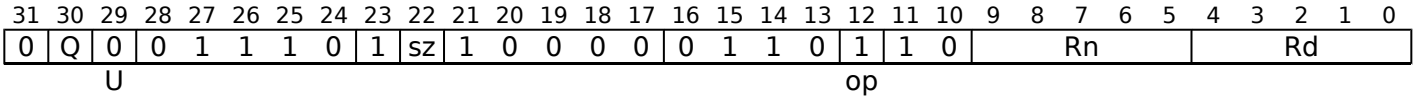
Vector half precision
(FEAT_FP16)



FCMEQ <Vd>.<T>, <Vn>.<T>, #0.0

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector single-precision and double-precision



FCMEQ <Vd>.<T>, <Vn>.<T>, #0.0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) zero = FPZero('0', esize);
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR[]);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR[]);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR[]);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR[]);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR[]);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCMGE (register)

Floating-point Compare Greater than or Equal (vector). This instruction reads each floating-point value in the first source SIMD&FP register and if the value is greater than or equal to the corresponding floating-point value in the second source SIMD&FP register sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

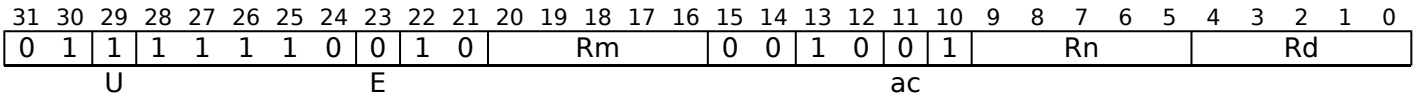
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

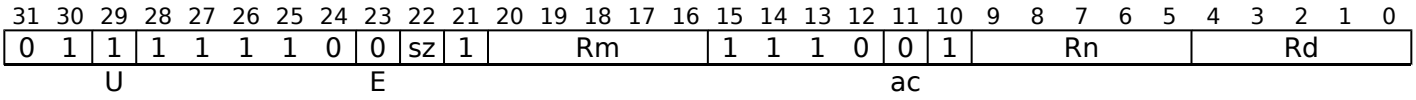
(FEAT_FP16)



FCMGE <Hd>, <Hn>, <Hm>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

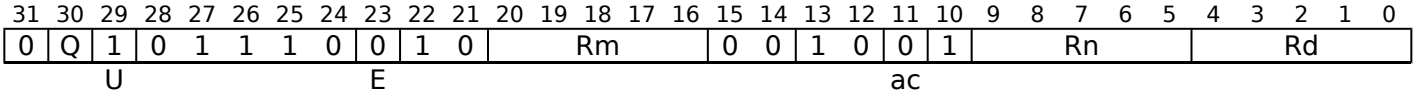
Scalar single-precision and double-precision



FCMGE <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

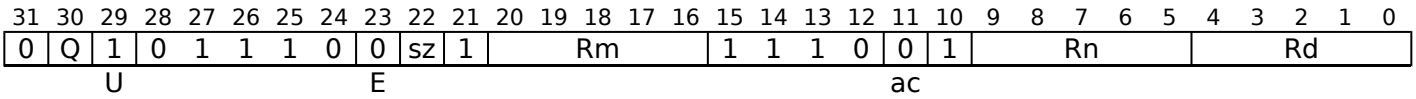
Vector half precision
(FEAT_FP16)



FCMGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Vector single-precision and double-precision



FCMGE <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
boolean test_passed;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[m, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, fpcr);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, fpcr);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, fpcr);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCMGE (zero)

Floating-point Compare Greater than or Equal to zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is greater than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

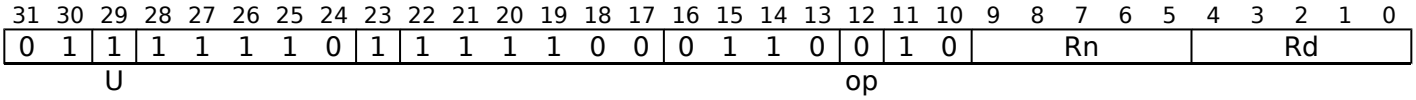
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

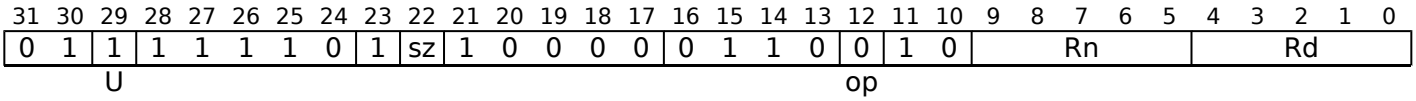
(FEAT_FP16)



FCMGE <Hd>, <Hn>, #0.0

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

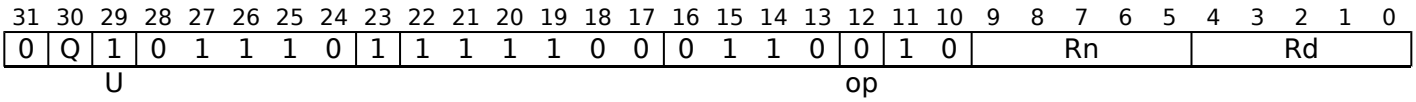
Scalar single-precision and double-precision



FCMGE <V><d>, <V><n>, #0.0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

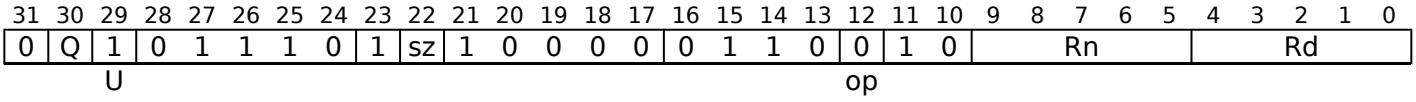
Vector half precision
(FEAT_FP16)



FCMGE <Vd>.<T>, <Vn>.<T>, #0.0

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector single-precision and double-precision



FCMGE <Vd>.<T>, <Vn>.<T>, #0.0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) zero = FPZero('0', esize);
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR[]);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR[]);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR[]);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR[]);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR[]);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCMGT (register)

Floating-point Compare Greater than (vector). This instruction reads each floating-point value in the first source SIMD&FP register and if the value is greater than the corresponding floating-point value in the second source SIMD&FP register sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

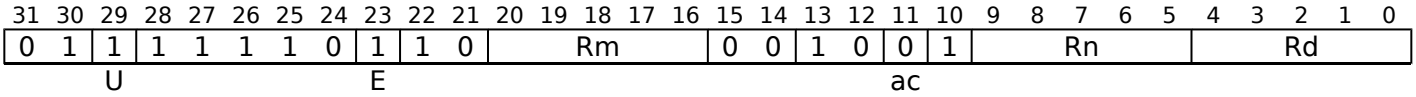
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

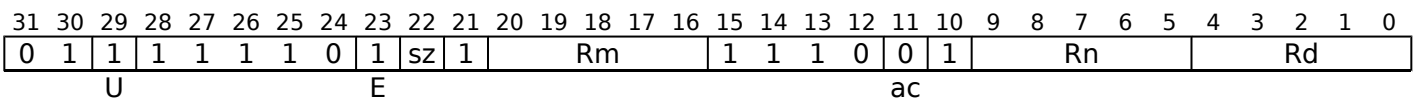
(FEAT_FP16)



FCMGT <Hd>, <Hn>, <Hm>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

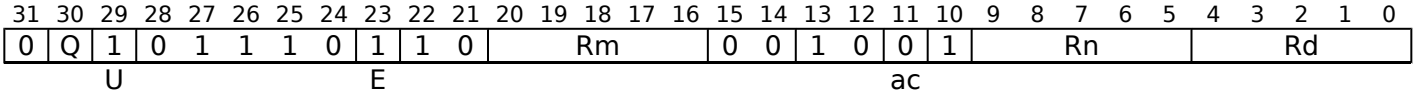
Scalar single-precision and double-precision



FCMGT <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

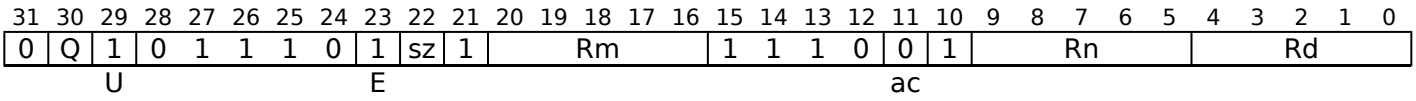
Vector half precision
(FEAT_FP16)



FCMGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Vector single-precision and double-precision



FCMGT <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
CompareOp cmp;
boolean abs;

case E:U:ac of
    when '000' cmp = CompareOp_EQ; abs = FALSE;
    when '010' cmp = CompareOp_GE; abs = FALSE;
    when '011' cmp = CompareOp_GE; abs = TRUE;
    when '110' cmp = CompareOp_GT; abs = FALSE;
    when '111' cmp = CompareOp_GT; abs = TRUE;
    otherwise UNDEFINED;
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
boolean test_passed;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[m, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if abs then
        element1 = FPAbs(element1);
        element2 = FPAbs(element2);
    case cmp of
        when CompareOp_EQ test_passed = FPCompareEQ(element1, element2, fpcr);
        when CompareOp_GE test_passed = FPCompareGE(element1, element2, fpcr);
        when CompareOp_GT test_passed = FPCompareGT(element1, element2, fpcr);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCMGT (zero)

Floating-point Compare Greater than zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is greater than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

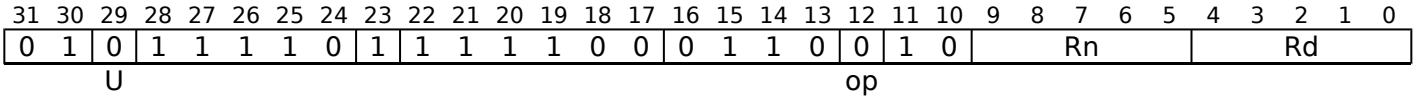
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

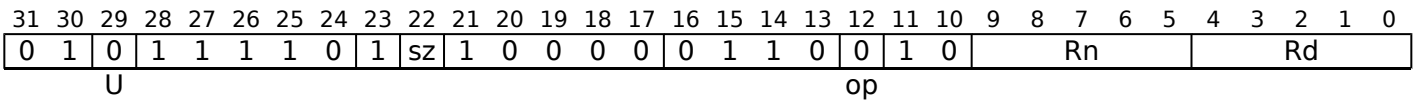
(FEAT_FP16)



FCMGT <Hd>, <Hn>, #0.0

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

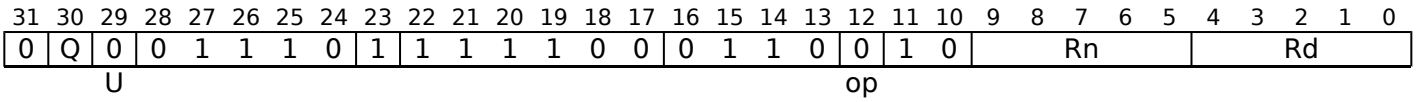
Scalar single-precision and double-precision



FCMGT <V><d>, <V><n>, #0.0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

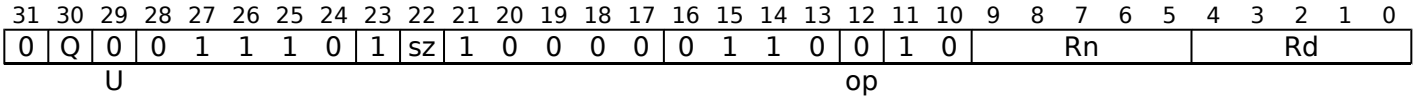
Vector half precision
(FEAT_FP16)



FCMGT <Vd>.<T>, <Vn>.<T>, #0.0

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector single-precision and double-precision



FCMGT <Vd>.<T>, <Vn>.<T>, #0.0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) zero = FPZero('0', esize);
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR[]);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR[]);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR[]);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR[]);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR[]);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCMLA

Floating-point Complex Multiply Accumulate.

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on the corresponding complex number element pairs from the two source registers and the destination register:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination register.

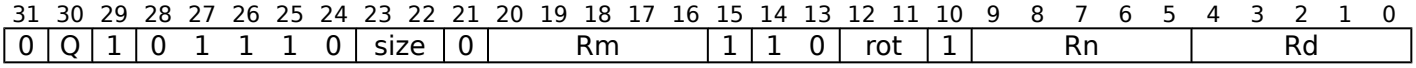
The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_FCMA)



FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<rotate>

```
if !HaveFCADDExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '00' then UNDEFINED;
if Q == '0' && size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<rotate> Is the rotation, encoded in “rot”:

rot	<rotate>
00	0
01	90
10	180
11	270

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) element3;
bits(esize) element4;
FPCRType fpcr = FPCR[];

for e = 0 to (elements DIV 2)-1
    case rot of
        when '00'
            element1 = Elem[operand2, e*2, esize];
            element2 = Elem[operand1, e*2, esize];
            element3 = Elem[operand2, e*2+1, esize];
            element4 = Elem[operand1, e*2, esize];
        when '01'
            element1 = FPNeg(Elem[operand2, e*2+1, esize]);
            element2 = Elem[operand1, e*2+1, esize];
            element3 = Elem[operand2, e*2, esize];
            element4 = Elem[operand1, e*2+1, esize];
        when '10'
            element1 = FPNeg(Elem[operand2, e*2, esize]);
            element2 = Elem[operand1, e*2, esize];
            element3 = FPNeg(Elem[operand2, e*2+1, esize]);
            element4 = Elem[operand1, e*2, esize];
        when '11'
            element1 = Elem[operand2, e*2+1, esize];
            element2 = Elem[operand1, e*2+1, esize];
            element3 = FPNeg(Elem[operand2, e*2, esize]);
            element4 = Elem[operand1, e*2+1, esize];

    Elem[result, e*2, esize] = FPMulAdd(Elem[operand3, e*2, esize], element2, element1, fpcr);
    Elem[result, e*2+1, esize] = FPMulAdd(Elem[operand3, e*2+1, esize], element4, element3, fpcr);

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCMLA (by element)

Floating-point Complex Multiply Accumulate (by element).

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on complex numbers from the first source register and the destination register with the specified complex number from the second source register:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector

(FEAT_FCMA)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	size		L	M	Rm				0	rot		1	H	0	Rn					Rd				

(size == 01)

```
FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>], #<rotate>
```

(size == 10)

```
FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>], #<rotate>
```

```
if !HaveFCADDExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer index;
if size == '00' || size == '11' then UNDEFINED;
if size == '01' then index = UInt(H:L);
if size == '10' then index = UInt(H);
integer esize = 8 << UInt(size);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
if size == '10' && (L == '1' || Q == '0') then UNDEFINED;
if size == '01' && H == '1' && Q == '0' then UNDEFINED;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:H:L”:

size	<index>
00	RESERVED
01	H:L
10	H
11	RESERVED

<rotate> Is the rotation, encoded in “rot”:

rot	<rotate>
00	0
01	90
10	180
11	270

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
FPCRType fpcr = FPCR[];

for e = 0 to (elements DIV 2)-1
    bits(esize) element1;
    bits(esize) element2;
    bits(esize) element3;
    bits(esize) element4;
    case rot of
        when '00'
            element1 = Elem[operand2, index*2, esize];
            element2 = Elem[operand1, e*2, esize];
            element3 = Elem[operand2, index*2+1, esize];
            element4 = Elem[operand1, e*2, esize];
        when '01'
            element1 = FPNeg(Elem[operand2, index*2+1, esize]);
            element2 = Elem[operand1, e*2+1, esize];
            element3 = Elem[operand2, index*2, esize];
            element4 = Elem[operand1, e*2+1, esize];
        when '10'
            element1 = FPNeg(Elem[operand2, index*2, esize]);
            element2 = Elem[operand1, e*2, esize];
            element3 = FPNeg(Elem[operand2, index*2+1, esize]);
            element4 = Elem[operand1, e*2, esize];
        when '11'
            element1 = Elem[operand2, index*2+1, esize];
            element2 = Elem[operand1, e*2+1, esize];
            element3 = FPNeg(Elem[operand2, index*2, esize]);
            element4 = Elem[operand1, e*2+1, esize];

    Elem[result, e*2, esize] = FPMulAdd(Elem[operand3, e*2, esize], element2, element1, fpcr);
    Elem[result, e*2+1, esize] = FPMulAdd(Elem[operand3, e*2+1, esize], element4, element3, fpcr);

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCMLE (zero)

Floating-point Compare Less than or Equal to zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is less than or equal to zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

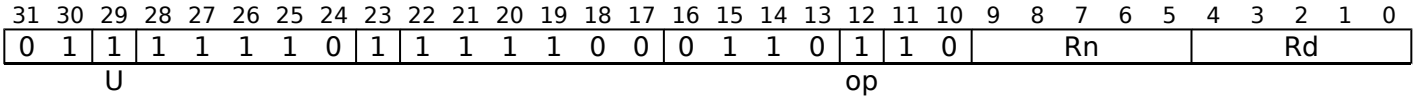
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

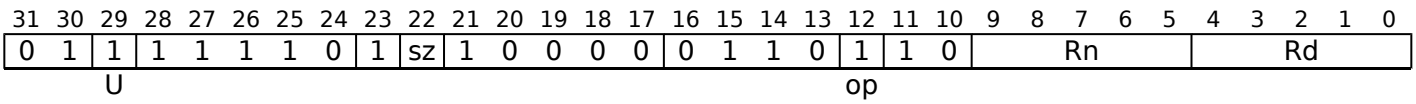
(FEAT_FP16)



FCMLE <Hd>, <Hn>, #0.0

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

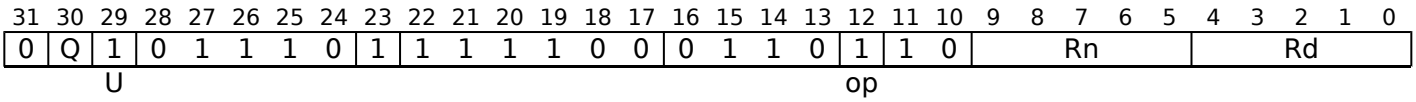
Scalar single-precision and double-precision



FCMLE <V><d>, <V><n>, #0.0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

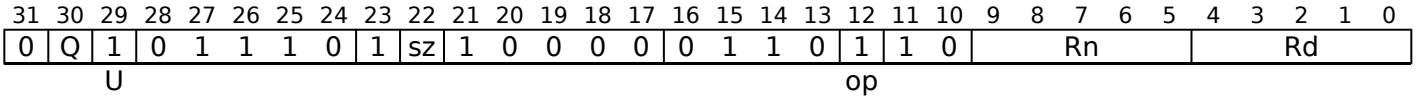
Vector half precision
(FEAT_FP16)



FCMLE <Vd>.<T>, <Vn>.<T>, #0.0

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Vector single-precision and double-precision



FCMLE <Vd>.<T>, <Vn>.<T>, #0.0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison;
case op:U of
    when '00' comparison = CompareOp_GT;
    when '01' comparison = CompareOp_GE;
    when '10' comparison = CompareOp_EQ;
    when '11' comparison = CompareOp_LE;
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) zero = FPZero('0', esize);
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR[]);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR[]);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR[]);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR[]);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR[]);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCMLT (zero)

Floating-point Compare Less than zero (vector). This instruction reads each floating-point value in the source SIMD&FP register and if the value is less than zero sets every bit of the corresponding vector element in the destination SIMD&FP register to one, otherwise sets every bit of the corresponding vector element in the destination SIMD&FP register to zero.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	1	1	1	1	0	0	0	1	1	1	0	1	0	Rn					Rd				

FCMLT <Hd>, <Hn>, #0.0

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

CompareOp comparison = CompareOp_LT;
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	sz	1	0	0	0	0	0	1	1	1	0	1	0	Rn					Rd				

FCMLT <V><d>, <V><n>, #0.0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

CompareOp comparison = CompareOp_LT;
```

Vector half precision

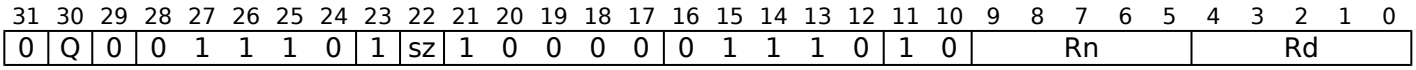
(FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	1	1	1	0	0	0	1	1	1	0	1	0	Rn					Rd				

FCMLT <Vd>.<T>, <Vn>.<T>, #0.0

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison = CompareOp_LT;
```

Vector single-precision and double-precision



FCMLT <Vd>.<T>, <Vn>.<T>, #0.0

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

CompareOp comparison = CompareOp_LT;
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) zero = FPZero('0', esize);
bits(esize) element;
boolean test_passed;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    case comparison of
        when CompareOp_GT test_passed = FPCompareGT(element, zero, FPCR[]);
        when CompareOp_GE test_passed = FPCompareGE(element, zero, FPCR[]);
        when CompareOp_EQ test_passed = FPCompareEQ(element, zero, FPCR[]);
        when CompareOp_LE test_passed = FPCompareGE(zero, element, FPCR[]);
        when CompareOp_LT test_passed = FPCompareGT(zero, element, FPCR[]);
    Elem[result, e, esize] = if test_passed then Ones(esize) else Zeros(esize);

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

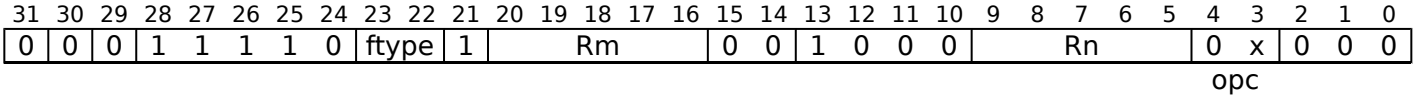
FCMP

Floating-point quiet Compare (scalar). This instruction compares the two SIMD&FP source register values, or the first SIMD&FP source register value and zero. It writes the result to the *PSTATE*.{N, Z, C, V} flags.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is a signaling NaN.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11 && opc == 00)
(FEAT_FP16)

```
FCMP <Hn>, <Hm>
```

Half-precision, zero (ftype == 11 && Rm == (00000) && opc == 01)
(FEAT_FP16)

```
FCMP <Hn>, #0.0
```

Single-precision (ftype == 00 && opc == 00)

```
FCMP <Sn>, <Sm>
```

Single-precision, zero (ftype == 00 && Rm == (00000) && opc == 01)

```
FCMP <Sn>, #0.0
```

Double-precision (ftype == 01 && opc == 00)

```
FCMP <Dn>, <Dm>
```

Double-precision, zero (ftype == 01 && Rm == (00000) && opc == 01)

```
FCMP <Dn>, #0.0
```

```
integer n = UInt(Rn);
integer m = UInt(Rm);    // ignored when opc<0> == '1'

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;

boolean signal_all_nans = (opc<1> == '1');
boolean cmp_with_zero = (opc<0> == '1');
```

Assembler Symbols

<Dn>	For the double-precision variant: is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field. For the double-precision, zero variant: is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hn>	For the half-precision variant: is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field. For the half-precision, zero variant: is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sn>	For the single-precision variant: is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field. For the single-precision, zero variant: is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPEnabled64();

bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2;

operand2 = if cmp_with_zero then FPZero('0', datasize) else V[m, datasize];

PSTATE.<N,Z,C,V> = FPCompare(operand1, operand2, signal_all_nans, FPCR[]);
```

Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the *PSTATE* condition flags to N=0, Z=0, C=1, and V=1.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

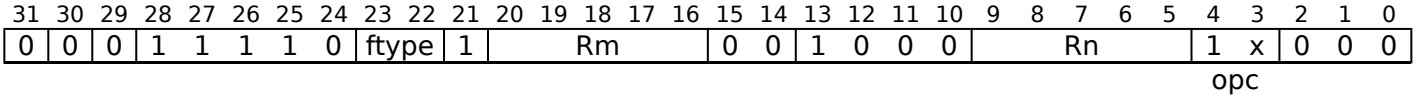
FCMPE

Floating-point signaling Compare (scalar). This instruction compares the two SIMD&FP source register values, or the first SIMD&FP source register value and zero. It writes the result to the *PSTATE*.{N, Z, C, V} flags.

This instruction raises an Invalid Operation floating-point exception if either or both of the operands is any type of NaN.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11 && opc == 10)
(FEAT_FP16)

```
FCMPE <Hn>, <Hm>
```

Half-precision, zero (ftype == 11 && Rm == (00000) && opc == 11)
(FEAT_FP16)

```
FCMPE <Hn>, #0.0
```

Single-precision (ftype == 00 && opc == 10)

```
FCMPE <Sn>, <Sm>
```

Single-precision, zero (ftype == 00 && Rm == (00000) && opc == 11)

```
FCMPE <Sn>, #0.0
```

Double-precision (ftype == 01 && opc == 10)

```
FCMPE <Dn>, <Dm>
```

Double-precision, zero (ftype == 01 && Rm == (00000) && opc == 11)

```
FCMPE <Dn>, #0.0
```

```
integer n = UInt(Rn);
integer m = UInt(Rm);    // ignored when opc<0> == '1'

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;

boolean signal_all_nans = (opc<1> == '1');
boolean cmp_with_zero = (opc<0> == '1');
```

Assembler Symbols

<Dn>	For the double-precision variant: is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field. For the double-precision, zero variant: is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hn>	For the half-precision variant: is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field. For the half-precision, zero variant: is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sn>	For the single-precision variant: is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field. For the single-precision, zero variant: is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPEnabled64();

bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2;

operand2 = if cmp_with_zero then FPZero('0', datasize) else V[m, datasize];

PSTATE.<N,Z,C,V> = FPCompare(operand1, operand2, signal_all_nans, FPCR[]);
```

Operational information

The IEEE 754 standard specifies that the result of a comparison is precisely one of <, ==, > or unordered. If either or both of the operands is a NaN, they are unordered, and all three of (Operand1 < Operand2), (Operand1 == Operand2) and (Operand1 > Operand2) are false. An unordered comparison sets the *PSTATE* condition flags to N=0, Z=0, C=1, and V=1.

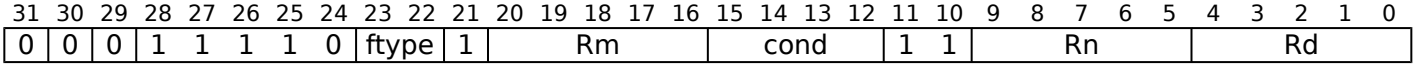
If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCSEL

Floating-point Conditional Select (scalar). This instruction allows the SIMD&FP destination register to take the value from either one or the other of two SIMD&FP source registers. If the condition passes, the first SIMD&FP source register value is taken, otherwise the second SIMD&FP source register value is taken.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)

(FEAT_FP16)

```
FCSEL <Hd>, <Hn>, <Hm>, <cond>
```

Single-precision (ftype == 00)

```
FCSEL <Sd>, <Sn>, <Sm>, <cond>
```

Double-precision (ftype == 01)

```
FCSEL <Dd>, <Dn>, <Dm>, <cond>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<cond>	Is one of the standard conditions, encoded in the "cond" field in the standard way.

Operation

```
CheckFPEnabled64();
bits(datasize) result;

result = if ConditionHolds(cond) then V[n, datasize] else V[m, datasize];

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

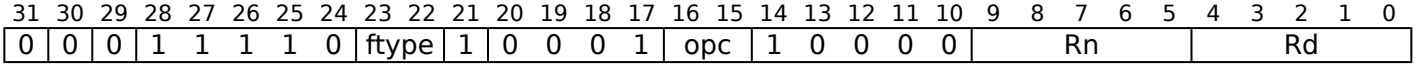
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCVT

Floating-point Convert precision (scalar). This instruction converts the floating-point value in the SIMD&FP source register to the precision for the destination register data type using the rounding mode that is determined by the *FPCR* and writes the result to the SIMD&FP destination register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision to single-precision (ftype == 11 && opc == 00)

```
FCVT <Sd>, <Hn>
```

Half-precision to double-precision (ftype == 11 && opc == 01)

```
FCVT <Dd>, <Hn>
```

Single-precision to half-precision (ftype == 00 && opc == 11)

```
FCVT <Hd>, <Sn>
```

Single-precision to double-precision (ftype == 00 && opc == 01)

```
FCVT <Dd>, <Sn>
```

Double-precision to half-precision (ftype == 01 && opc == 11)

```
FCVT <Hd>, <Dn>
```

Double-precision to single-precision (ftype == 01 && opc == 00)

```
FCVT <Sd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer srcsize;
integer dstsize;

if ftype == opc then UNDEFINED;

case ftype of
    when '00' srcsize = 32;
    when '01' srcsize = 64;
    when '10' UNDEFINED;
    when '11' srcsize = 16;
case opc of
    when '00' dstsize = 32;
    when '01' dstsize = 64;
    when '10' UNDEFINED;
    when '11' dstsize = 16;
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEnabled64();

bits(srcsize) operand = V[n, srcsize];
FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

Elem[result, 0, dstsize] = FPConvert(operand, fpcr, dstsize);

V[d, 128] = result;
```

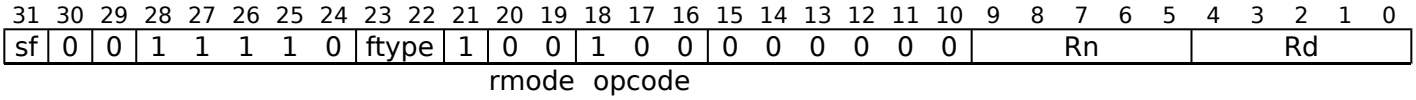
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCVTAS (scalar)

Floating-point Convert to Signed integer, rounding to nearest with ties to Away (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round to Nearest with Ties to Away rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision to 32-bit (sf == 0 && ftype == 11)
(FEAT_FP16)

```
FCVTAS <Wd>, <Hn>
```

Half-precision to 64-bit (sf == 1 && ftype == 11)
(FEAT_FP16)

```
FCVTAS <Xd>, <Hn>
```

Single-precision to 32-bit (sf == 0 && ftype == 00)

```
FCVTAS <Wd>, <Sn>
```

Single-precision to 64-bit (sf == 1 && ftype == 00)

```
FCVTAS <Xd>, <Sn>
```

Double-precision to 32-bit (sf == 0 && ftype == 01)

```
FCVTAS <Wd>, <Dn>
```

Double-precision to 64-bit (sf == 1 && ftype == 01)

```
FCVTAS <Xd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n, fltsize];
intval = FPToFixed(fltval, 0, FALSE, fpcr, FPRounding_TIEAWAY, intsize);
X[d, intsize] = intval;
```

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCVTAS (vector)

Floating-point Convert to Signed integer, rounding to nearest with ties to Away (vector). This instruction converts each element in a vector from a floating-point value to a signed integer value using the Round to Nearest with Ties to Away rounding mode and writes the result to the SIMD&FP destination register.

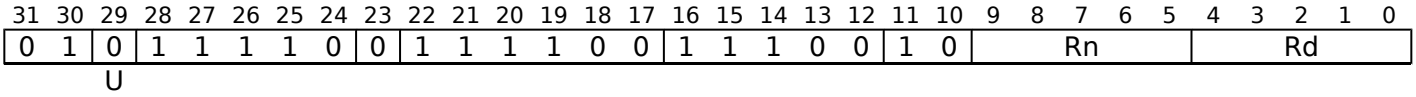
A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

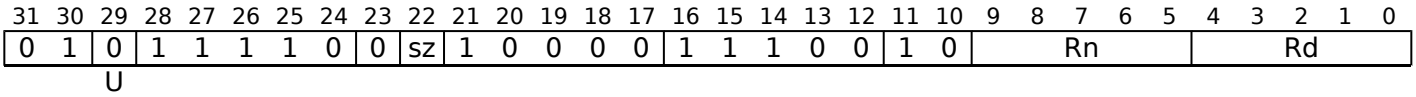
(FEAT_FP16)



FCVTAS <Hd>, <Hn>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

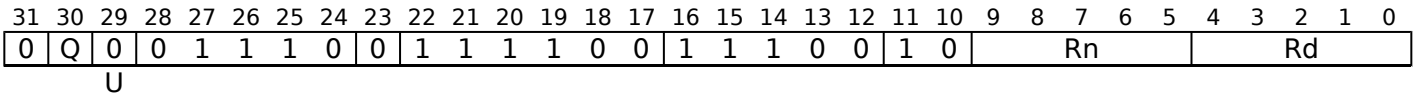


FCVTAS <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Vector half precision

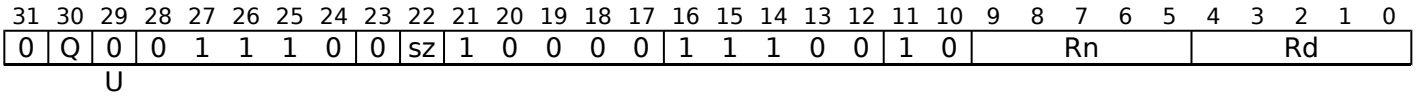
(FEAT_FP16)



FCVTAS <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



FCVTAS <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding, esize);

V[d, 128] = result;
```

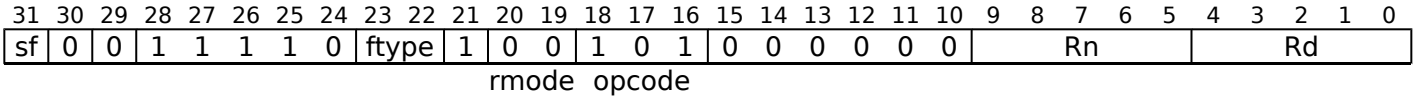
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCVTAU (scalar)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round to Nearest with Ties to Away rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision to 32-bit (sf == 0 && ftype == 11)
(FEAT_FP16)

```
FCVTAU <Wd>, <Hn>
```

Half-precision to 64-bit (sf == 1 && ftype == 11)
(FEAT_FP16)

```
FCVTAU <Xd>, <Hn>
```

Single-precision to 32-bit (sf == 0 && ftype == 00)

```
FCVTAU <Wd>, <Sn>
```

Single-precision to 64-bit (sf == 1 && ftype == 00)

```
FCVTAU <Xd>, <Sn>
```

Double-precision to 32-bit (sf == 0 && ftype == 01)

```
FCVTAU <Wd>, <Dn>
```

Double-precision to 64-bit (sf == 1 && ftype == 01)

```
FCVTAU <Xd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n, fltsize];
intval = FPToFixed(fltval, 0, TRUE, fpcr, FPRounding_TIEAWAY, intsize);
X[d, intsize] = intval;
```

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCVTAU (vector)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to Away (vector). This instruction converts each element in a vector from a floating-point value to an unsigned integer value using the Round to Nearest with Ties to Away rounding mode and writes the result to the SIMD&FP destination register.

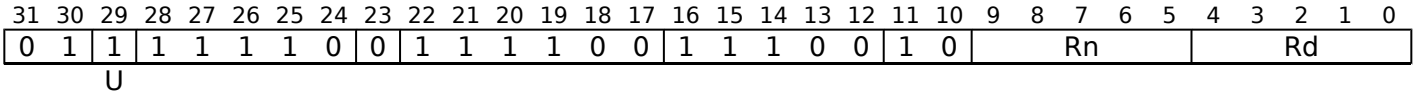
A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

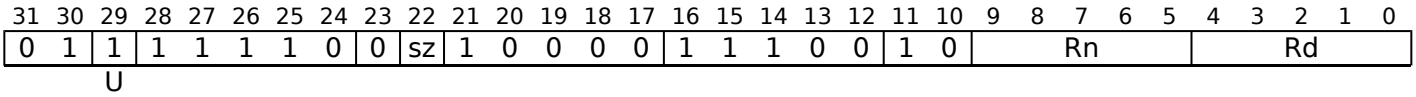
(FEAT_FP16)



FCVTAU <Hd>, <Hn>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

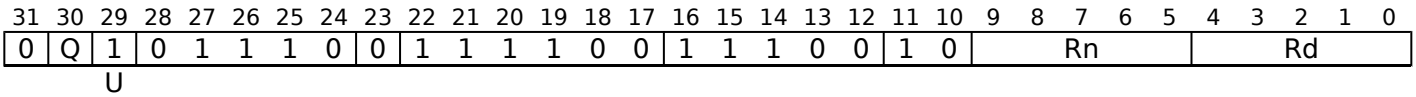


FCVTAU <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Vector half precision

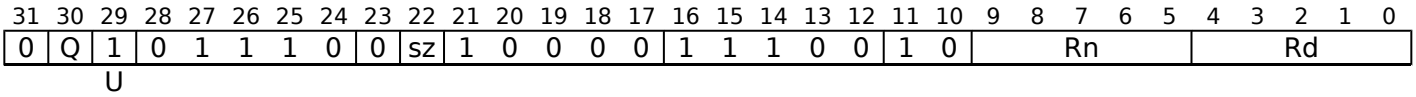
(FEAT_FP16)



FCVTAU <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



FCVTAU <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPRounding_TIEAWAY;
boolean unsigned = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding, esize);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

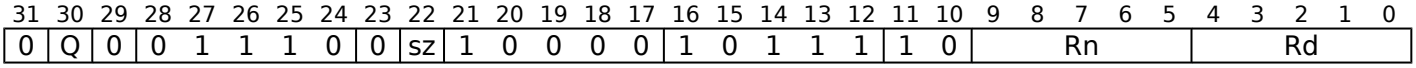
FCVTL, FCVTL2

Floating-point Convert to higher precision Long (vector). This instruction reads each element in a vector in the SIMD&FP source register, converts each value to double the precision of the source element using the rounding mode that is determined by the *FPCR*, and writes each result to the equivalent element of the vector in the SIMD&FP destination register.

Where the operation lengthens a 64-bit vector to a 128-bit vector, the FCVTL2 variant operates on the elements in the top 64 bits of the source register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



FCVTL{2} <Vd>.<Ta>, <Vn>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16 << UInt(sz);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “sz”:

sz	<Ta>
0	4S
1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<Tb>
0	0	4H
0	1	8H
1	0	2S
1	1	4S

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part, datasize];
bits(2*datasize) result;

for e = 0 to elements-1
    Elem[result, e, 2*esize] = FPConvert(Elem[operand, e, esize], FPCR[], 2 * esize);

V[d, 2*datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCVTMS (scalar)

Floating-point Convert to Signed integer, rounding toward Minus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Minus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	0	1	1	1	1	0	ftype		1	1	0	0	0	0	0	0	0	0	0	0	Rn					Rd				
											rmode		opcode																			

Half-precision to 32-bit (sf == 0 && ftype == 11)
(FEAT_FP16)

```
FCVTMS <Wd>, <Hn>
```

Half-precision to 64-bit (sf == 1 && ftype == 11)
(FEAT_FP16)

```
FCVTMS <Xd>, <Hn>
```

Single-precision to 32-bit (sf == 0 && ftype == 00)

```
FCVTMS <Wd>, <Sn>
```

Single-precision to 64-bit (sf == 1 && ftype == 00)

```
FCVTMS <Xd>, <Sn>
```

Double-precision to 32-bit (sf == 0 && ftype == 01)

```
FCVTMS <Wd>, <Dn>
```

Double-precision to 64-bit (sf == 1 && ftype == 01)

```
FCVTMS <Xd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

rounding = FPDecodeRounding(rmode);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n, fltsize];
intval = FPToFixed(fltval, 0, FALSE, fpcr, rounding, intsize);
X[d, intsize] = intval;
```

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCVTMS (vector)

Floating-point Convert to Signed integer, rounding toward Minus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

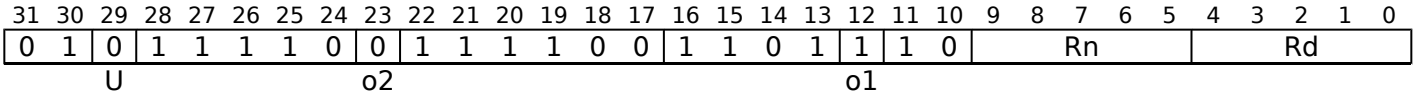
A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

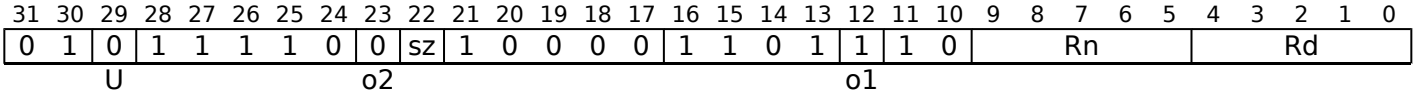
(FEAT_FP16)



FCVTMS <Hd>, <Hn>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

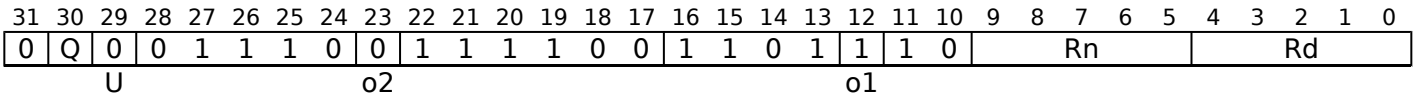


FCVTMS <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector half precision

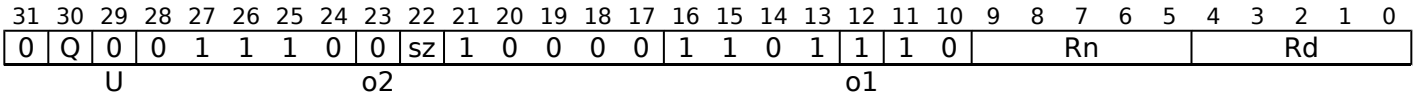
(FEAT_FP16)



FCVTMS <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



FCVTMS <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding, esize);

V[d, 128] = result;
```

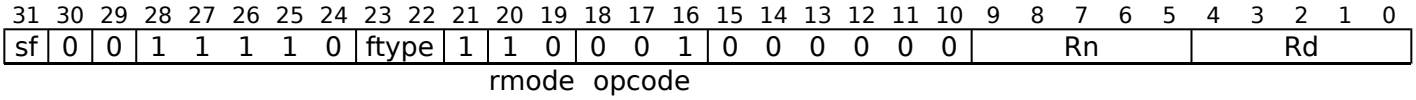
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCVTMU (scalar)

Floating-point Convert to Unsigned integer, rounding toward Minus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Minus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision to 32-bit (sf == 0 && ftype == 11)
(FEAT_FP16)

```
FCVTMU <Wd>, <Hn>
```

Half-precision to 64-bit (sf == 1 && ftype == 11)
(FEAT_FP16)

```
FCVTMU <Xd>, <Hn>
```

Single-precision to 32-bit (sf == 0 && ftype == 00)

```
FCVTMU <Wd>, <Sn>
```

Single-precision to 64-bit (sf == 1 && ftype == 00)

```
FCVTMU <Xd>, <Sn>
```

Double-precision to 32-bit (sf == 0 && ftype == 01)

```
FCVTMU <Wd>, <Dn>
```

Double-precision to 64-bit (sf == 1 && ftype == 01)

```
FCVTMU <Xd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

rounding = FPDecodeRounding(rmode);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n, fltsize];
intval = FPToFixed(fltval, 0, TRUE, fpcr, rounding, intsize);
X[d, intsize] = intval;
```

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCVTMU (vector)

Floating-point Convert to Unsigned integer, rounding toward Minus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

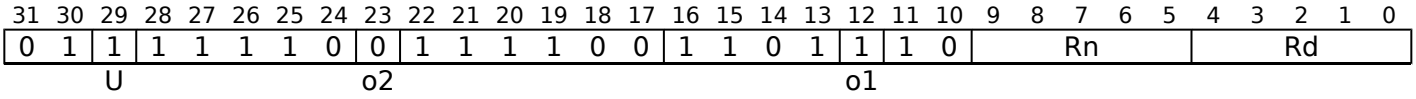
A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

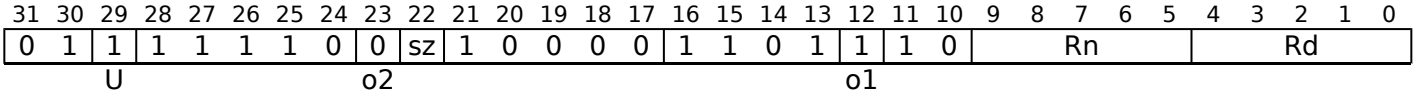
(FEAT_FP16)



FCVTMU <Hd>, <Hn>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

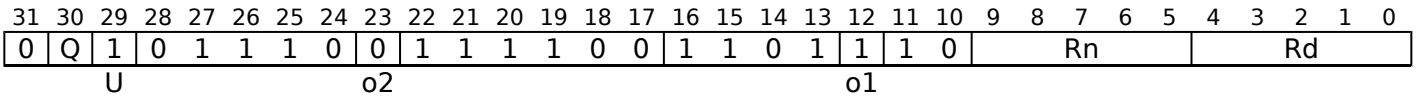


FCVTMU <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector half precision

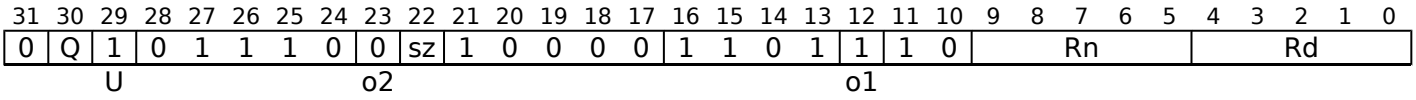
(FEAT_FP16)



FCVTMU <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



FCVTMU <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding, esize);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

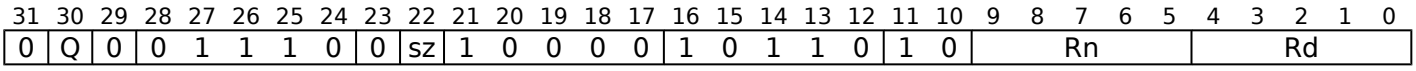
FCVTN, FCVTN2

Floating-point Convert to lower precision Narrow (vector). This instruction reads each vector element in the SIMD&FP source register, converts each result to half the precision of the source element, writes the final result to a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The rounding mode is determined by the *FPCR*.

The FCVTN instruction writes the vector to the lower half of the destination register and clears the upper half, while the FCVTN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.



FCVTN{2} <Vd>.<Tb>, <Vn>.<Ta>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16 << UInt(sz);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<Tb>
0	0	4H
0	1	8H
1	0	2S
1	1	4S

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “sz”:

sz	<Ta>
0	4S
1	2D

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n, 2*datasize];
bits(datasize) result;

for e = 0 to elements-1
    Elem[result, e, esize] = FPConvert(Elem[operand, e, 2*esize], FPCR[], esize);

Vpart[d, part, datasize] = result;
```

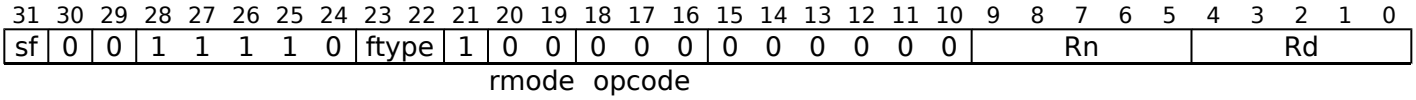
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCVTNS (scalar)

Floating-point Convert to Signed integer, rounding to nearest with ties to even (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round to Nearest rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision to 32-bit (sf == 0 && ftype == 11)
(FEAT_FP16)

```
FCVTNS <Wd>, <Hn>
```

Half-precision to 64-bit (sf == 1 && ftype == 11)
(FEAT_FP16)

```
FCVTNS <Xd>, <Hn>
```

Single-precision to 32-bit (sf == 0 && ftype == 00)

```
FCVTNS <Wd>, <Sn>
```

Single-precision to 64-bit (sf == 1 && ftype == 00)

```
FCVTNS <Xd>, <Sn>
```

Double-precision to 32-bit (sf == 0 && ftype == 01)

```
FCVTNS <Wd>, <Dn>
```

Double-precision to 64-bit (sf == 1 && ftype == 01)

```
FCVTNS <Xd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

rounding = FPDecodeRounding(rmode);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n, fltsize];
intval = FPToFixed(fltval, 0, FALSE, fpcr, rounding, intsize);
X[d, intsize] = intval;
```

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCVTNS (vector)

Floating-point Convert to Signed integer, rounding to nearest with ties to even (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

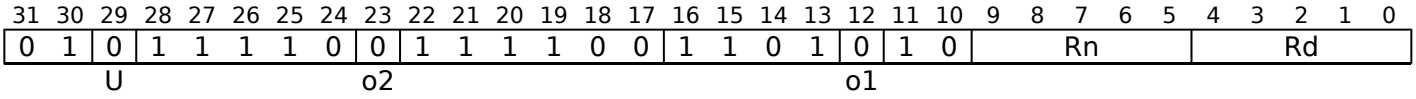
A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

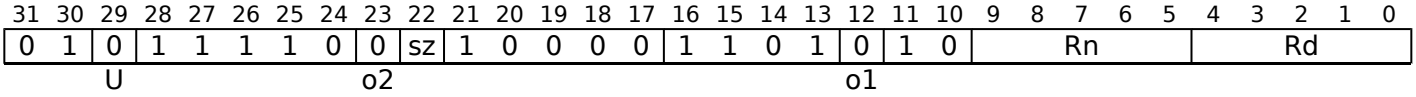
(FEAT_FP16)



FCVTNS <Hd>, <Hn>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

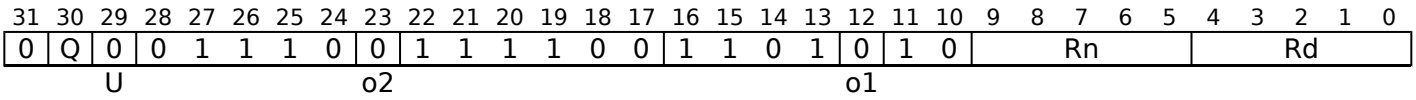


FCVTNS <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector half precision

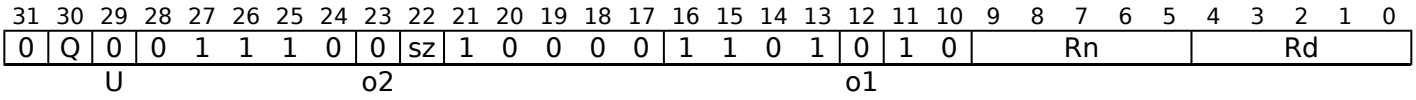
(FEAT_FP16)



FCVTNS <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



FCVTNS <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding, esize);

V[d, 128] = result;
```

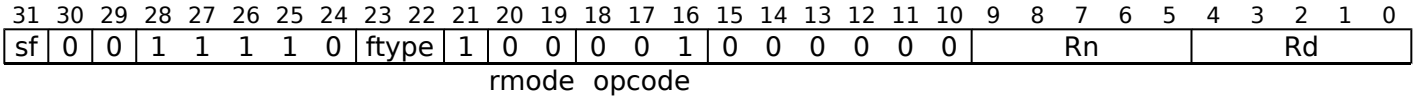
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCVTNU (scalar)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round to Nearest rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision to 32-bit (sf == 0 && ftype == 11)
(FEAT_FP16)

```
FCVTNU <Wd>, <Hn>
```

Half-precision to 64-bit (sf == 1 && ftype == 11)
(FEAT_FP16)

```
FCVTNU <Xd>, <Hn>
```

Single-precision to 32-bit (sf == 0 && ftype == 00)

```
FCVTNU <Wd>, <Sn>
```

Single-precision to 64-bit (sf == 1 && ftype == 00)

```
FCVTNU <Xd>, <Sn>
```

Double-precision to 32-bit (sf == 0 && ftype == 01)

```
FCVTNU <Wd>, <Dn>
```

Double-precision to 64-bit (sf == 1 && ftype == 01)

```
FCVTNU <Xd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

rounding = FPDecodeRounding(rmode);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n, fltsize];
intval = FPToFixed(fltval, 0, TRUE, fpcr, rounding, intsize);
X[d, intsize] = intval;
```

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCVTNU (vector)

Floating-point Convert to Unsigned integer, rounding to nearest with ties to even (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

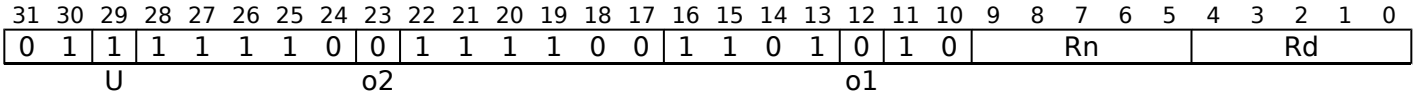
A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

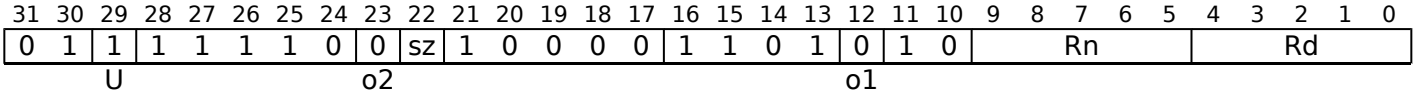
(FEAT_FP16)



FCVTNU <Hd>, <Hn>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

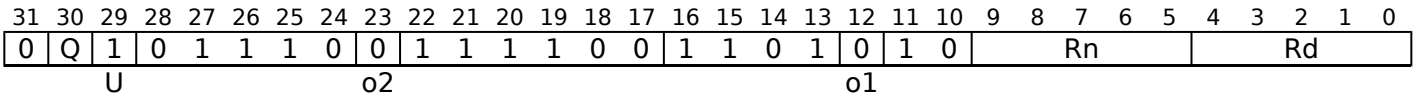


FCVTNU <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector half precision

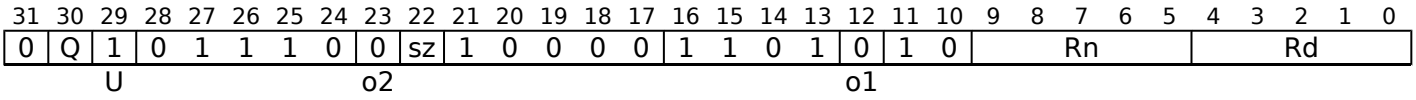
(FEAT_FP16)



FCVTNU <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



FCVTNU <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding, esize);

V[d, 128] = result;
```

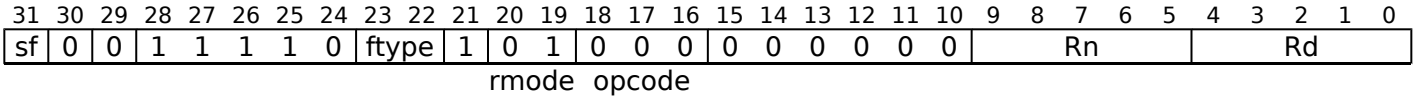
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCVTPS (scalar)

Floating-point Convert to Signed integer, rounding toward Plus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Plus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision to 32-bit (sf == 0 && ftype == 11)
(FEAT_FP16)

```
FCVTPS <Wd>, <Hn>
```

Half-precision to 64-bit (sf == 1 && ftype == 11)
(FEAT_FP16)

```
FCVTPS <Xd>, <Hn>
```

Single-precision to 32-bit (sf == 0 && ftype == 00)

```
FCVTPS <Wd>, <Sn>
```

Single-precision to 64-bit (sf == 1 && ftype == 00)

```
FCVTPS <Xd>, <Sn>
```

Double-precision to 32-bit (sf == 0 && ftype == 01)

```
FCVTPS <Wd>, <Dn>
```

Double-precision to 64-bit (sf == 1 && ftype == 01)

```
FCVTPS <Xd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

rounding = FPDecodeRounding(rmode);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n, fltsize];
intval = FPToFixed(fltval, 0, FALSE, fpcr, rounding, intsize);
X[d, intsize] = intval;
```

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCVTPS (vector)

Floating-point Convert to Signed integer, rounding toward Plus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

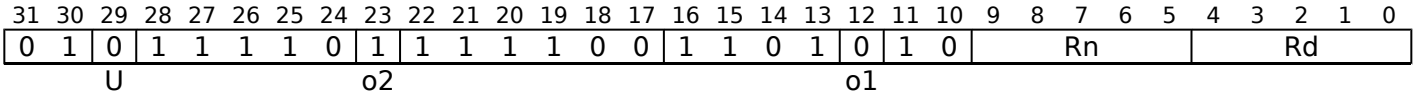
A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

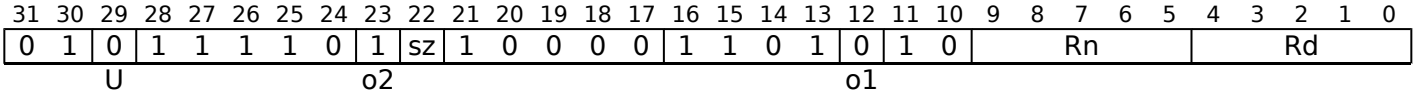
(FEAT_FP16)



FCVTPS <Hd>, <Hn>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

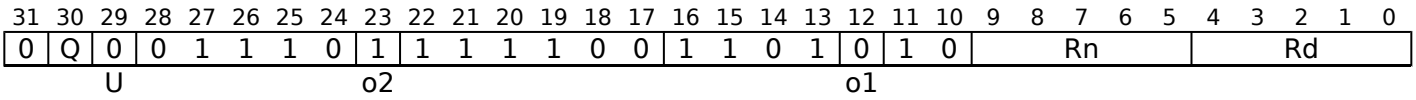


FCVTPS <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector half precision

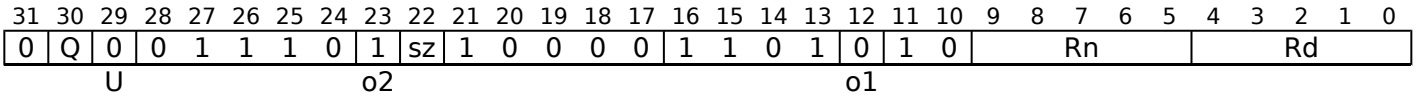
(FEAT_FP16)



FCVTPS <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



FCVTPS <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding, esize);

V[d, 128] = result;
```

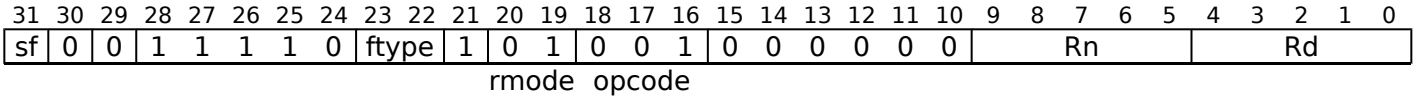
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCVTPU (scalar)

Floating-point Convert to Unsigned integer, rounding toward Plus infinity (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Plus Infinity rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision to 32-bit (sf == 0 && ftype == 11)
(FEAT_FP16)

```
FCVTPU <Wd>, <Hn>
```

Half-precision to 64-bit (sf == 1 && ftype == 11)
(FEAT_FP16)

```
FCVTPU <Xd>, <Hn>
```

Single-precision to 32-bit (sf == 0 && ftype == 00)

```
FCVTPU <Wd>, <Sn>
```

Single-precision to 64-bit (sf == 1 && ftype == 00)

```
FCVTPU <Xd>, <Sn>
```

Double-precision to 32-bit (sf == 0 && ftype == 01)

```
FCVTPU <Wd>, <Dn>
```

Double-precision to 64-bit (sf == 1 && ftype == 01)

```
FCVTPU <Xd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

rounding = FPDecodeRounding(rmode);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n, fltsize];
intval = FPToFixed(fltval, 0, TRUE, fpcr, rounding, intsize);
X[d, intsize] = intval;
```

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCVTPU (vector)

Floating-point Convert to Unsigned integer, rounding toward Plus infinity (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

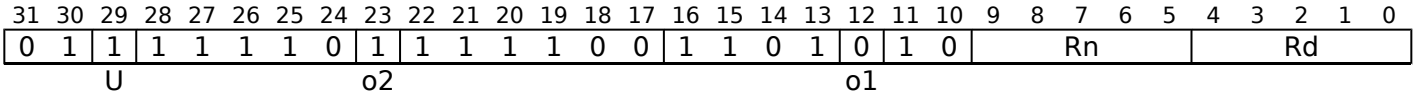
A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

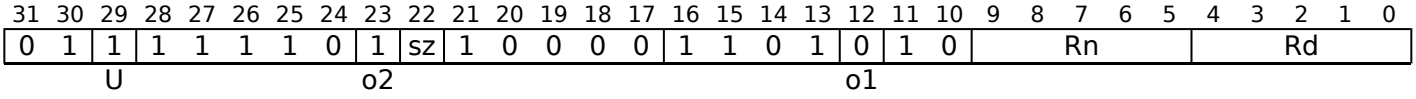
(FEAT_FP16)



FCVTPU <Hd>, <Hn>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

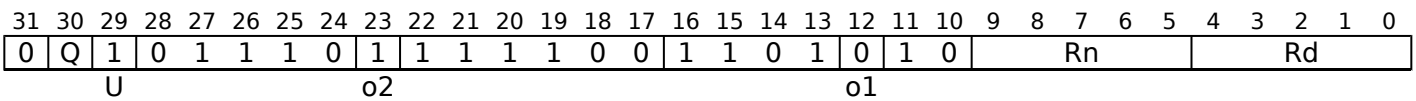


FCVTPU <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector half precision

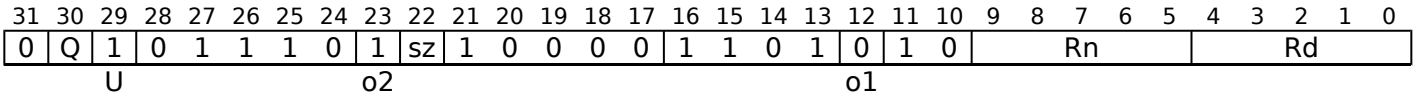
(FEAT_FP16)



FCVTPU <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



FCVTPU <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding, esize);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCVTXN, FCVTXN2

Floating-point Convert to lower precision Narrow, rounding to odd (vector). This instruction reads each vector element in the source SIMD&FP register, narrows each value to half the precision of the source element using the Round to Odd rounding mode, writes the result to a vector, and writes the vector to the destination SIMD&FP register.

Note

This instruction uses the Round to Odd rounding mode which is not defined by the IEEE 754-2008 standard. This rounding mode ensures that if the result of the conversion is inexact the least significant bit of the mantissa is forced to 1. This rounding mode enables a floating-point value to be converted to a lower precision format via an intermediate precision format while avoiding double rounding errors. For example, a 64-bit floating-point value can be converted to a correctly rounded 16-bit floating-point value by first using this instruction to produce a 32-bit value and then using another instruction with the wanted rounding mode to convert the 32-bit value to the final 16-bit floating-point value.

The FCVTXN instruction writes the vector to the lower half of the destination register and clears the upper half, while the FCVTXN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	0	sz	1	0	0	0	0	1	0	1	1	0	1	0	Rn					Rd				

FCVTXN <Vb><d>, <Va><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '0' then UNDEFINED;
integer esize = 32;
integer datasize = esize;
integer elements = 1;
integer part = 0;
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	0	0	0	0	1	0	1	1	0	1	0	Rn					Rd				

FCVTXN{2} <Vd>.<Tb>, <Vn>.<Ta>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '0' then UNDEFINED;
integer esize = 32;
integer datasize = 64;
integer elements = 2;
integer part = UInt(Q);
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<Tb>
0	x	RESERVED
1	0	2S
1	1	4S

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “sz”:

sz	<Ta>
0	RESERVED
1	2D

<Vb> Is the destination width specifier, encoded in “sz”:

sz	<Vb>
0	RESERVED
1	S

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va> Is the source width specifier, encoded in “sz”:

sz	<Va>
0	RESERVED
1	D

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();

bits(2*datasize) operand = V[n, 2*datasize];
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
    Elem[result, e, esize] = FPConvert(Elem[operand, e, 2*esize], fpcr, FPRounding_ODD, esize);

if merge then
    V[d, 128] = result;
else
    Vpart[d, part, datasize] = Elem[result, 0, datasize];
```

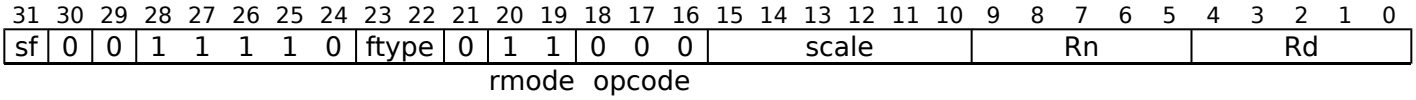
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCVTZS (scalar, fixed-point)

Floating-point Convert to Signed fixed-point, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit fixed-point signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.



Half-precision to 32-bit (sf == 0 && ftype == 11)
(FEAT_FP16)

```
FCVTZS <Wd>, <Hn>, #<fbits>
```

Half-precision to 64-bit (sf == 1 && ftype == 11)
(FEAT_FP16)

```
FCVTZS <Xd>, <Hn>, #<fbits>
```

Single-precision to 32-bit (sf == 0 && ftype == 00)

```
FCVTZS <Wd>, <Sn>, #<fbits>
```

Single-precision to 64-bit (sf == 1 && ftype == 00)

```
FCVTZS <Xd>, <Sn>, #<fbits>
```

Double-precision to 32-bit (sf == 0 && ftype == 01)

```
FCVTZS <Wd>, <Dn>, #<fbits>
```

Double-precision to 64-bit (sf == 1 && ftype == 01)

```
FCVTZS <Xd>, <Dn>, #<fbits>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case ftype of
    when '00' fltsize = 32;
    when '01' fltsize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<fbits>	For the double-precision to 32-bit, half-precision to 32-bit and single-precision to 32-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 32, encoded as 64 minus "scale". For the double-precision to 64-bit, half-precision to 64-bit and single-precision to 64-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 64, encoded as 64 minus "scale".

Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n, fltsize];
intval = FPToFixed(fltval, fracbits, FALSE, fpcr, FPRounding_ZERO, intsize);
X[d, intsize] = intval;
```

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

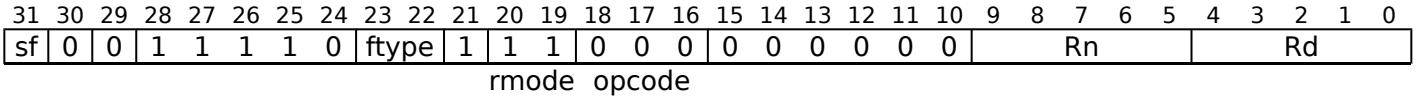
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCVTZS (scalar, integer)

Floating-point Convert to Signed integer, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision to 32-bit (sf == 0 && ftype == 11)
(FEAT_FP16)

```
FCVTZS <Wd>, <Hn>
```

Half-precision to 64-bit (sf == 1 && ftype == 11)
(FEAT_FP16)

```
FCVTZS <Xd>, <Hn>
```

Single-precision to 32-bit (sf == 0 && ftype == 00)

```
FCVTZS <Wd>, <Sn>
```

Single-precision to 64-bit (sf == 1 && ftype == 00)

```
FCVTZS <Xd>, <Sn>
```

Double-precision to 32-bit (sf == 0 && ftype == 01)

```
FCVTZS <Wd>, <Dn>
```

Double-precision to 64-bit (sf == 1 && ftype == 01)

```
FCVTZS <Xd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

rounding = FPDecodeRounding(rmode);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n, fltsize];
intval = FPToFixed(fltval, 0, FALSE, fpcr, rounding, intsize);
X[d, intsize] = intval;
```

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCVTZS (vector, fixed-point)

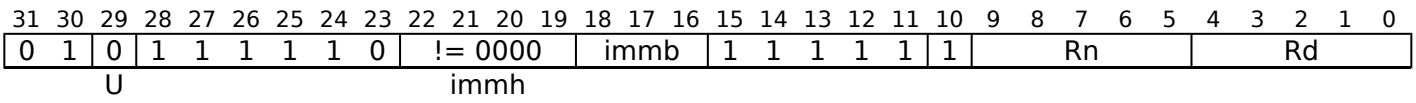
Floating-point Convert to Signed fixed-point, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from floating-point to fixed-point signed integer using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

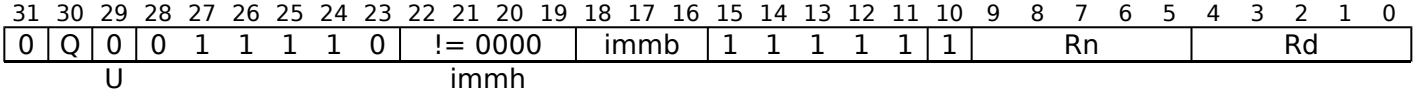
Scalar



FCVTZS <V><d>, <V><n>, #<fbits>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh IN {'000x'} || (immh IN {'001x'} && !HaveFP16Ext()) then UNDEFINED;
integer esize = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;
```

Vector



FCVTZS <Vd>.<T>, <Vn>.<T>, #<fbits>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh IN {'000x'} || (immh IN {'001x'} && !HaveFP16Ext()) then UNDEFINED;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
000x	RESERVED
001x	H
01xx	S
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	x	RESERVED
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in “immh:immb”:

immh	<fbits>
000x	RESERVED
001x	(32-Uint(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in “immh:immb”:

immh	<fbits>
0000	SEE Advanced SIMD modified immediate
0001	RESERVED
001x	(32-Uint(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, fpcr, rounding, esize);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCVTZS (vector, integer)

Floating-point Convert to Signed integer, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from a floating-point value to a signed integer value using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

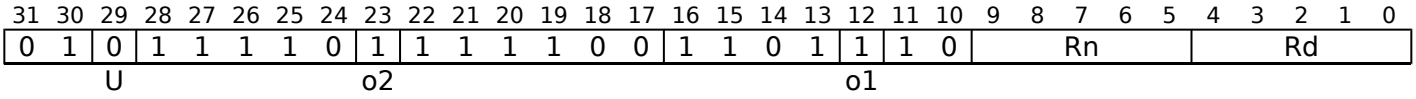
A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

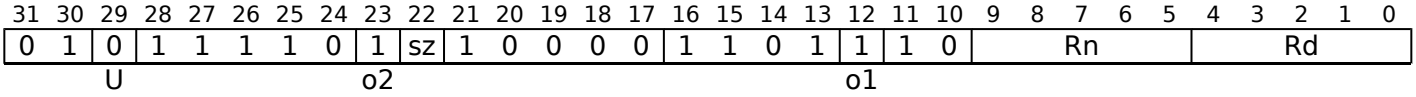
(FEAT_FP16)



FCVTZS <Hd>, <Hn>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

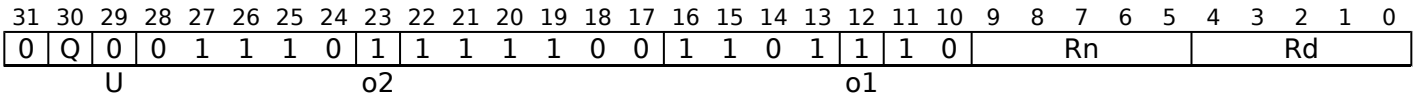


FCVTZS <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector half precision

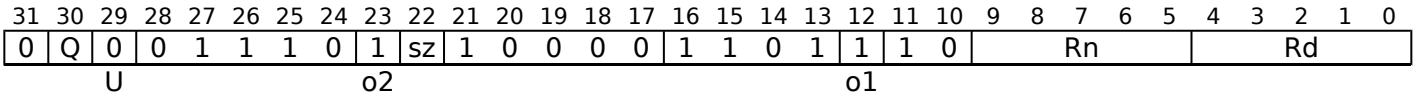
(FEAT_FP16)



FCVTZS <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



FCVTZS <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding, esize);

V[d, 128] = result;
```

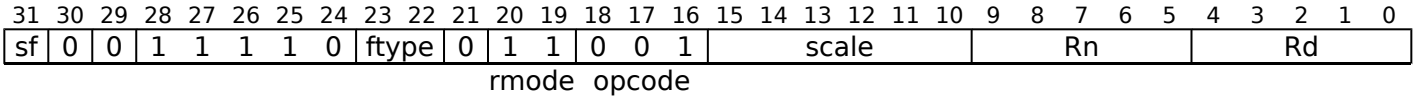
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCVTZU (scalar, fixed-point)

Floating-point Convert to Unsigned fixed-point, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit fixed-point unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.



Half-precision to 32-bit (sf == 0 && ftype == 11)
(FEAT_FP16)

```
FCVTZU <Wd>, <Hn>, #<fbits>
```

Half-precision to 64-bit (sf == 1 && ftype == 11)
(FEAT_FP16)

```
FCVTZU <Xd>, <Hn>, #<fbits>
```

Single-precision to 32-bit (sf == 0 && ftype == 00)

```
FCVTZU <Wd>, <Sn>, #<fbits>
```

Single-precision to 64-bit (sf == 1 && ftype == 00)

```
FCVTZU <Xd>, <Sn>, #<fbits>
```

Double-precision to 32-bit (sf == 0 && ftype == 01)

```
FCVTZU <Wd>, <Dn>, #<fbits>
```

Double-precision to 64-bit (sf == 1 && ftype == 01)

```
FCVTZU <Xd>, <Dn>, #<fbits>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;

case ftype of
    when '00' fltsize = 32;
    when '01' fltsize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<fbits>	For the double-precision to 32-bit, half-precision to 32-bit and single-precision to 32-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 32, encoded as 64 minus "scale". For the double-precision to 64-bit, half-precision to 64-bit and single-precision to 64-bit variant: is the number of bits after the binary point in the fixed-point destination, in the range 1 to 64, encoded as 64 minus "scale".

Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n, fltsize];
intval = FPToFixed(fltval, fracbits, TRUE, fpcr, FPRounding_ZERO, intsize);
X[d, intsize] = intval;
```

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

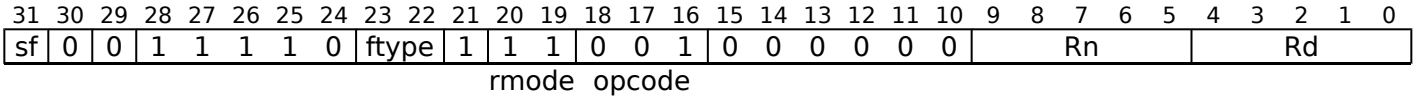
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCVTZU (scalar, integer)

Floating-point Convert to Unsigned integer, rounding toward Zero (scalar). This instruction converts the floating-point value in the SIMD&FP source register to a 32-bit or 64-bit unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision to 32-bit (sf == 0 && ftype == 11)
(FEAT_FP16)

```
FCVTZU <Wd>, <Hn>
```

Half-precision to 64-bit (sf == 1 && ftype == 11)
(FEAT_FP16)

```
FCVTZU <Xd>, <Hn>
```

Single-precision to 32-bit (sf == 0 && ftype == 00)

```
FCVTZU <Wd>, <Sn>
```

Single-precision to 64-bit (sf == 1 && ftype == 00)

```
FCVTZU <Xd>, <Sn>
```

Double-precision to 32-bit (sf == 0 && ftype == 01)

```
FCVTZU <Wd>, <Dn>
```

Double-precision to 64-bit (sf == 1 && ftype == 01)

```
FCVTZU <Xd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

rounding = FPDecodeRounding(rmode);
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
bits(fltsize) fltval;
bits(intsize) intval;

fltval = V[n, fltsize];
intval = FPToFixed(fltval, 0, TRUE, fpcr, rounding, intsize);
X[d, intsize] = intval;
```

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCVTZU (vector, fixed-point)

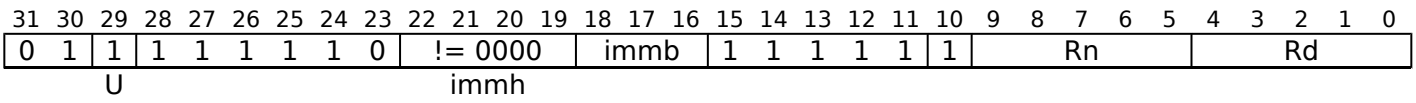
Floating-point Convert to Unsigned fixed-point, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from floating-point to fixed-point unsigned integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

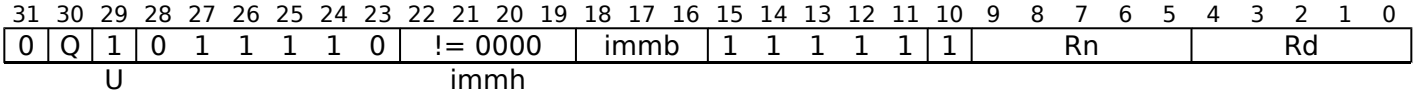
Scalar



FCVTZU <V><d>, <V><n>, #<fbits>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh IN {'000x'} || (immh IN {'001x'} && !HaveFP16Ext()) then UNDEFINED;
integer esize = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;
```

Vector



FCVTZU <Vd>.<T>, <Vn>.<T>, #<fbits>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh IN {'000x'} || (immh IN {'001x'} && !HaveFP16Ext()) then UNDEFINED;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRounding_ZERO;
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
000x	RESERVED
001x	H
01xx	S
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	x	RESERVED
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in “immh:immb”:

immh	<fbits>
000x	RESERVED
001x	(32-Uint(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in “immh:immb”:

immh	<fbits>
0000	SEE Advanced SIMD modified immediate
0001	RESERVED
001x	(32-Uint(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, fracbits, unsigned, fpcr, rounding, esize);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCVTZU (vector, integer)

Floating-point Convert to Unsigned integer, rounding toward Zero (vector). This instruction converts a scalar or each element in a vector from a floating-point value to an unsigned integer value using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

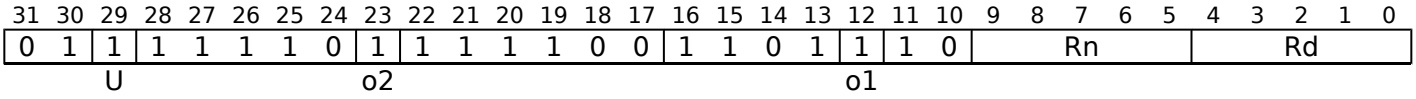
A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

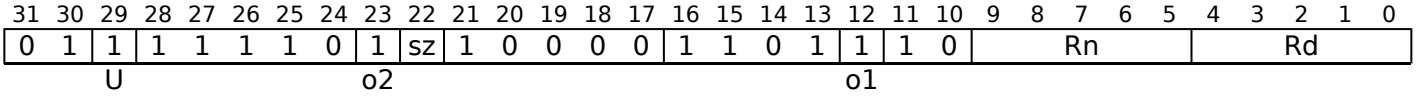
(FEAT_FP16)



FCVTZU <Hd>, <Hn>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

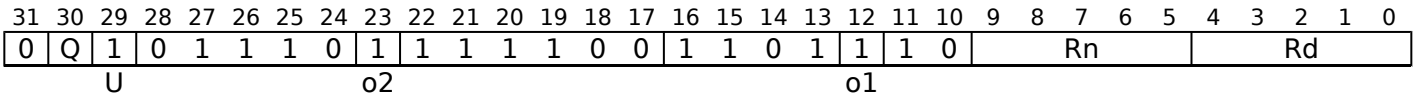


FCVTZU <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector half precision

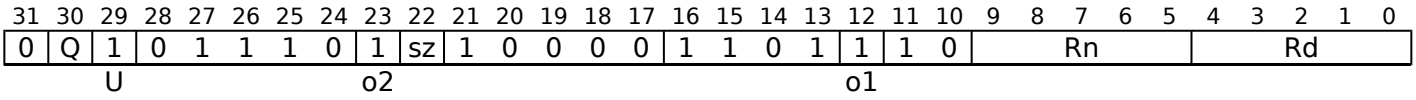
(FEAT_FP16)



FCVTZU <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



FCVTZU <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

FPRounding rounding = FPDecodeRounding(o1:o2);
boolean unsigned = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

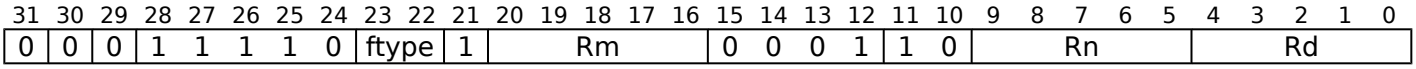
```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPToFixed(element, 0, unsigned, fpcr, rounding, esize);

V[d, 128] = result;
```

FDIV (scalar)

Floating-point Divide (scalar). This instruction divides the floating-point value of the first source SIMD&FP register by the floating-point value of the second source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)

(FEAT_FP16)

```
FDIV <Hd>, <Hn>, <Hm>
```

Single-precision (ftype == 00)

```
FDIV <Sd>, <Sn>, <Sm>
```

Double-precision (ftype == 01)

```
FDIV <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPEnabled64();
bits(esize) operand1 = V[n, esize];
bits(esize) operand2 = V[m, esize];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

Elem[result, 0, esize] = FPDiv(operand1, operand2, FPCR[]);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FDIV (vector)

Floating-point Divide (vector). This instruction divides the floating-point values in the elements in the first source SIMD&FP register, by the floating-point values in the corresponding elements in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

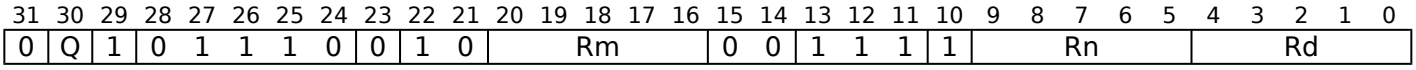
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

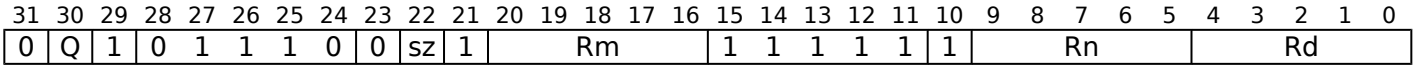
(FEAT_FP16)



FDIV <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Single-precision and double-precision



FDIV <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

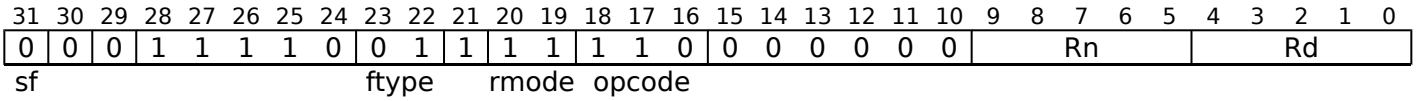
```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPDiv(element1, element2, FPCR[]);

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FJCVTZS

Floating-point Javascript Convert to Signed fixed-point, rounding toward Zero. This instruction converts the double-precision floating-point value in the SIMD&FP source register to a 32-bit signed integer using the Round towards Zero rounding mode, and writes the result to the general-purpose destination register. If the result is too large to be represented as a signed 32-bit integer, then the result is the integer modulo 2^{32}, as held in a 32-bit signed integer.
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.
Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Double-precision to 32-bit
(FEAT_JSCVT)



FJCVTZS <Wd>, <Dn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if !HaveFJCVTZSExt() then UNDEFINED;
```

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
FPCRType fpcr = FPCR[];
bits(64) fltval;
bits(32) intval;

bit z;
fltval = V[n, 64];
(intval, z) = FPToFixedJS(fltval, fpcr, TRUE, 32);
PSTATE.<N,Z,C,V> = '0':z:'00';
X[d, 32] = intval;
```

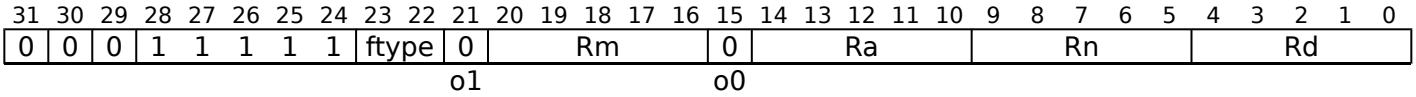
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMADD

Floating-point fused Multiply-Add (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, adds the product to the value of the third SIMD&FP source register, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)

(FEAT_FP16)

```
FMADD <Hd>, <Hn>, <Hm>, <Ha>
```

Single-precision (ftype == 00)

```
FMADD <Sd>, <Sn>, <Sm>, <Sa>
```

Double-precision (ftype == 01)

```
FMADD <Dd>, <Dn>, <Dm>, <Da>
```

```
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Da>	Is the 64-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Ha>	Is the 16-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

<Sa> Is the 32-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

Operation

```
CheckFPEnabled64();

bits(esize) operanda = V[a, esize];
bits(esize) operand1 = V[n, esize];
bits(esize) operand2 = V[m, esize];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[a, 128] else Zeros(128);

Elem[result, 0, esize] = FPMulAdd(operanda, operand1, operand2, fpcr);

V[d, 128] = result;
```

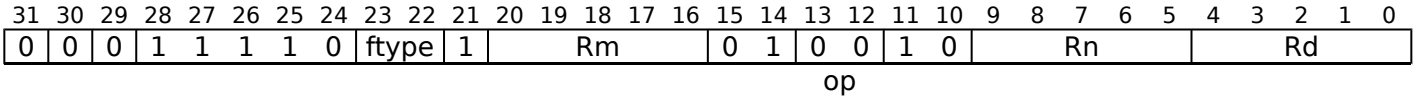
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMAX (scalar)

Floating-point Maximum (scalar). This instruction compares the two source SIMD&FP registers, and writes the larger of the two floating-point values to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)

(FEAT_FP16)

```
FMAX <Hd>, <Hn>, <Hm>
```

Single-precision (ftype == 00)

```
FMAX <Sd>, <Sn>, <Sm>
```

Double-precision (ftype == 01)

```
FMAX <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPEnabled64();
bits(esize) operand1 = V[n, esize];
bits(esize) operand2 = V[m, esize];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

Elem[result, 0, esize] = FPMax(operand1, operand2, fpcr);
V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMAX (vector)

Floating-point Maximum (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, places the larger of each of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

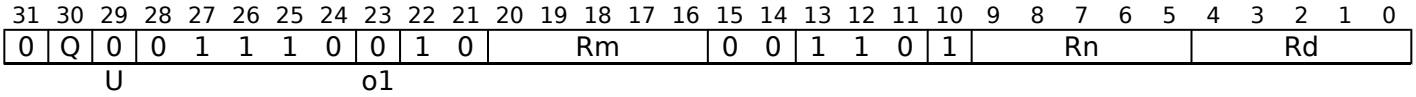
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

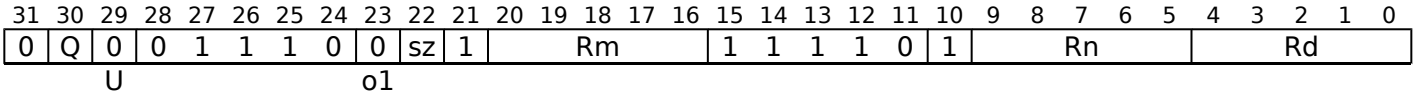
(FEAT_FP16)



FMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Single-precision and double-precision



FMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMin(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = FPMax(element1, element2, FPCR[]);

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

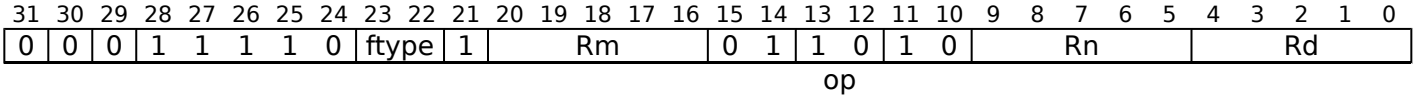
FMAXNM (scalar)

Floating-point Maximum Number (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the larger of the two floating-point values to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result that is placed in the vector is the numerical value, otherwise the result is identical to *FMAX (scalar)*.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)

(FEAT_FP16)

```
FMAXNM <Hd>, <Hn>, <Hm>
```

Single-precision (ftype == 00)

```
FMAXNM <Sd>, <Sn>, <Sm>
```

Double-precision (ftype == 01)

```
FMAXNM <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPEnabled64();
bits(esize) operand1 = V[n, esize];
bits(esize) operand2 = V[m, esize];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

Elem[result, 0, esize] = FPMaxNum(operand1, operand2, fpcr);
V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMAXNM (vector)

Floating-point Maximum Number (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, writes the larger of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result placed in the vector is the numerical value, otherwise the result is identical to *FMAX (scalar)*.

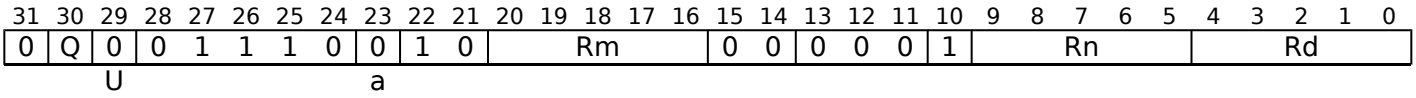
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

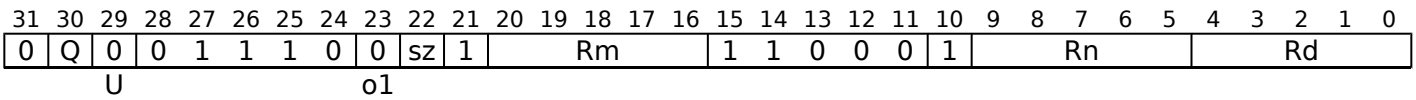
(FEAT_FP16)



FMAXNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');
```

Single-precision and double-precision



FMAXNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMinNum(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR[]);

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMAXNMP (scalar)

Floating-point Maximum Number of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the largest of the floating-point values as a scalar to the destination SIMD&FP register.

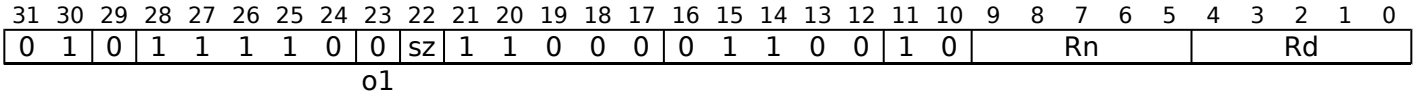
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

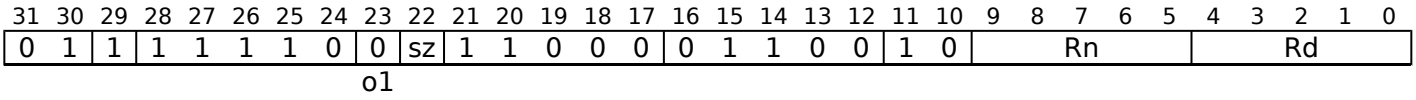
(FEAT_FP16)



FMAXNMP <V><d>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
if sz == '1' then UNDEFINED;
integer datasize = 32;
```

Single-precision and double-precision



FMAXNMP <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
```

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	H
1	RESERVED

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2H
1	RESERVED

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2S
1	2D

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
V[d, esize] = Reduce(ReduceOp_FMAXNUM, operand, esize, FALSE);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMAXNMP (vector)

Floating-point Maximum Number Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result is the numerical value, otherwise the result is identical to *FMAX (scalar)*.

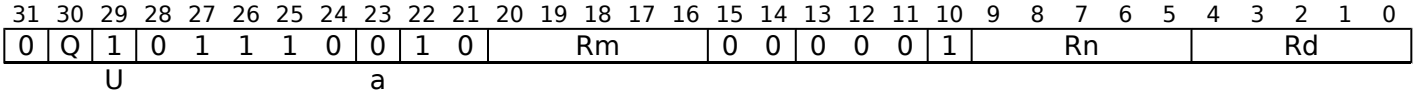
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

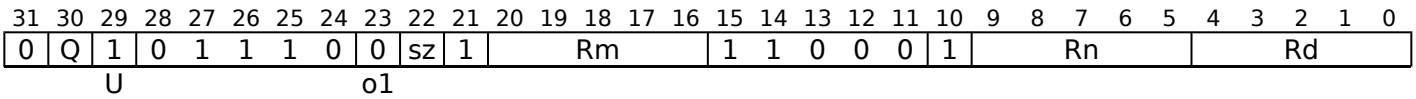
(FEAT_FP16)



FMAXNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');
```

Single-precision and double-precision



FMAXNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMinNum(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR[]);

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMAXNMV

Floating-point Maximum Number across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result of the comparison is the numerical value, otherwise the result is identical to *FMAX (scalar)*.

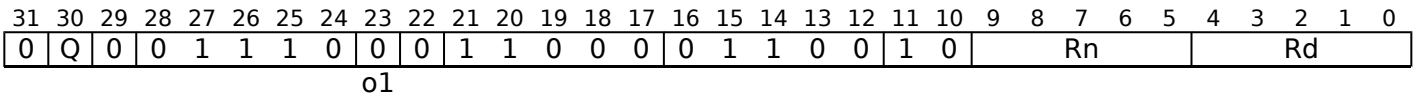
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

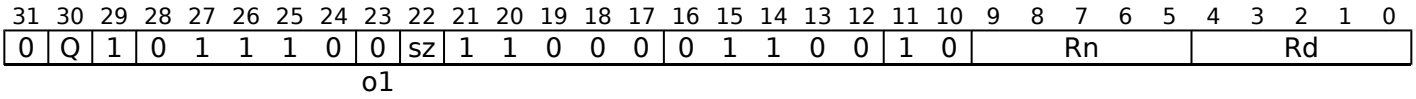
(FEAT_FP16)



FMAXNMV <V><d>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
```

Single-precision and double-precision



FMAXNMV <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then UNDEFINED;    // .4S only

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, H.

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	x	RESERVED
1	0	4S
1	1	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
V[d, esize] = Reduce(ReduceOp_FMAXNUM, operand, esize, FALSE);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMAXP (scalar)

Floating-point Maximum of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the largest of the floating-point values as a scalar to the destination SIMD&FP register.

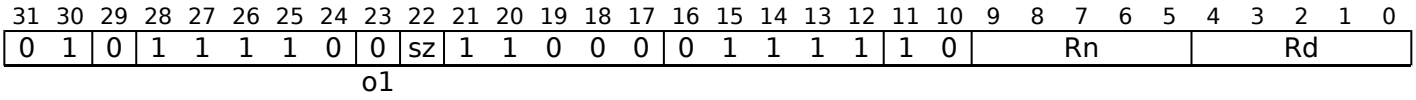
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

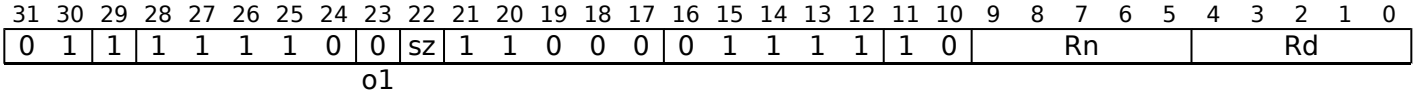
(FEAT_FP16)



FMAXP <V><d>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
if sz == '1' then UNDEFINED;
integer datasize = 32;
```

Single-precision and double-precision



FMAXP <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
```

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	H
1	RESERVED

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2H
1	RESERVED

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2S
1	2D

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

V[d, esize] = Reduce(ReduceOp_FMAX, operand, esize);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMAXP (vector)

Floating-point Maximum Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, writes the larger of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

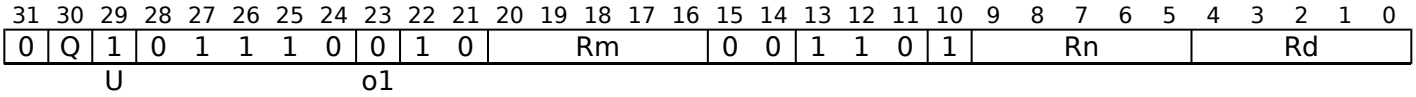
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

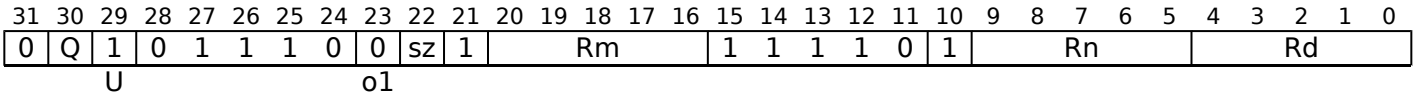
(FEAT_FP16)



FMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Single-precision and double-precision



FMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMin(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = FPMax(element1, element2, FPCR[]);

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMAXV

Floating-point Maximum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	0	1	1	0	0	0	0	1	1	1	1	1	0	Rn					Rd				
								o1																							

FMAXV <V><d>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	sz	1	1	0	0	0	0	1	1	1	1	1	0	Rn					Rd				
								o1																							

FMAXV <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then UNDEFINED;

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, H.

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	x	RESERVED
1	0	4S
1	1	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

V[d, esize] = Reduce(ReduceOp_FMAX, operand, esize);
```

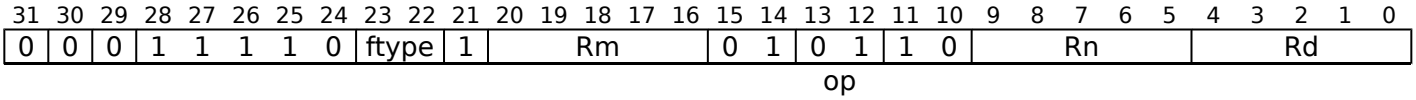
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMIN (scalar)

Floating-point Minimum (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the smaller of the two floating-point values to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)

(FEAT_FP16)

```
FMIN <Hd>, <Hn>, <Hm>
```

Single-precision (ftype == 00)

```
FMIN <Sd>, <Sn>, <Sm>
```

Double-precision (ftype == 01)

```
FMIN <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPEnabled64();
bits(esize) operand1 = V[n, esize];
bits(esize) operand2 = V[m, esize];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

Elem[result, 0, esize] = FPMin(operand1, operand2, fpcr);
V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMIN (vector)

Floating-point minimum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the smaller of each of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

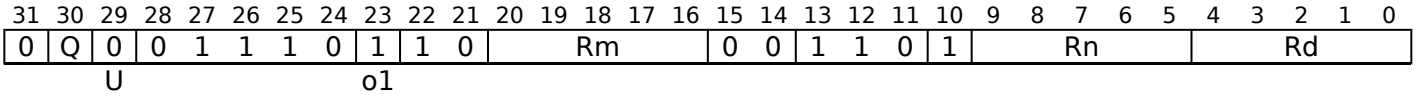
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

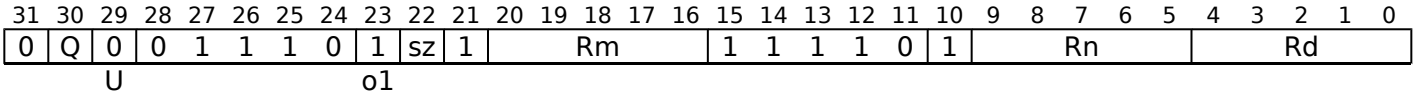
(FEAT_FP16)



FMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Single-precision and double-precision



FMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMin(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = FPMax(element1, element2, FPCR[]);

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

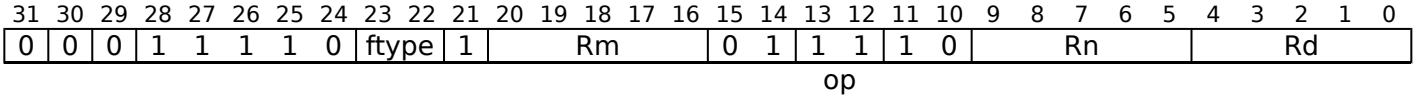
FMINNM (scalar)

Floating-point Minimum Number (scalar). This instruction compares the first and second source SIMD&FP register values, and writes the smaller of the two floating-point values to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result that is placed in the vector is the numerical value, otherwise the result is identical to *FMIN (scalar)*.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)

(FEAT_FP16)

```
FMINNM <Hd>, <Hn>, <Hm>
```

Single-precision (ftype == 00)

```
FMINNM <Sd>, <Sn>, <Sm>
```

Double-precision (ftype == 01)

```
FMINNM <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPEnabled64();
bits(esize) operand1 = V[n, esize];
bits(esize) operand2 = V[m, esize];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

Elem[result, 0, esize] = FPMinNum(operand1, operand2, fpcr);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMINNM (vector)

Floating-point Minimum Number (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, writes the smaller of the two floating-point values into a vector, and writes the vector to the destination SIMD&FP register.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result placed in the vector is the numerical value, otherwise the result is identical to *FMIN (scalar)*.

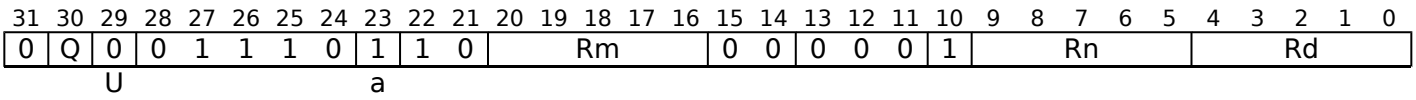
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

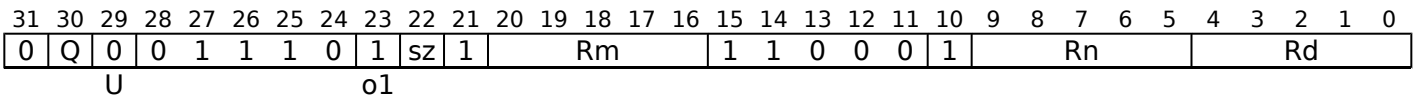
(FEAT_FP16)



FMINNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');
```

Single-precision and double-precision



FMINNM <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMinNum(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR[]);

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMINNMP (scalar)

Floating-point Minimum Number of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the smallest of the floating-point values as a scalar to the destination SIMD&FP register.

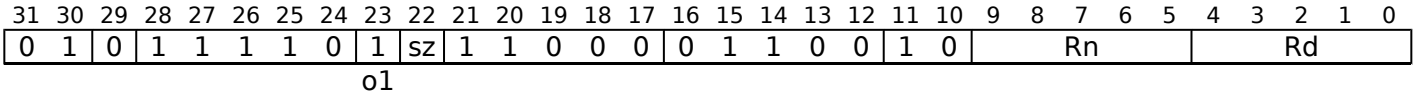
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

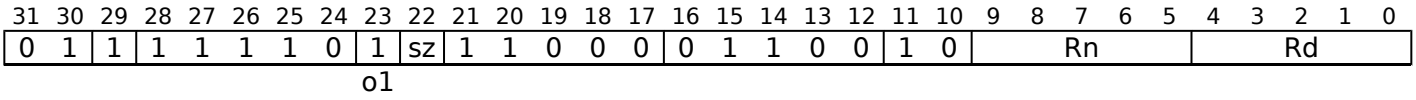
(FEAT_FP16)



FMINNMP <V><d>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
if sz == '1' then UNDEFINED;
integer datasize = 32;
```

Single-precision and double-precision



FMINNMP <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
```

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	H
1	RESERVED

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2H
1	RESERVED

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2S
1	2D

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
V[d, esize] = Reduce(ReduceOp_FMINNUM, operand, esize, FALSE);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMINNMP (vector)

Floating-point Minimum Number Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of floating-point values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result is the numerical value, otherwise the result is identical to *FMIN (scalar)*.

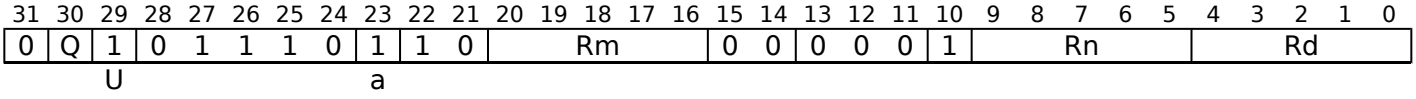
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

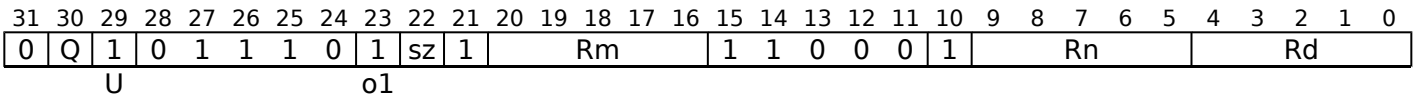
(FEAT_FP16)



FMINNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (a == '1');
```

Single-precision and double-precision



FMINNMP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMinNum(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR[]);

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMINNMV

Floating-point Minimum Number across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result of the comparison is the numerical value, otherwise the result is identical to *FMIN (scalar)*.

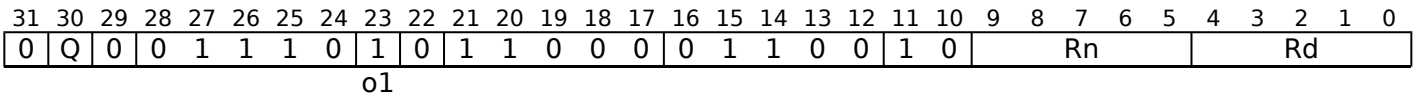
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

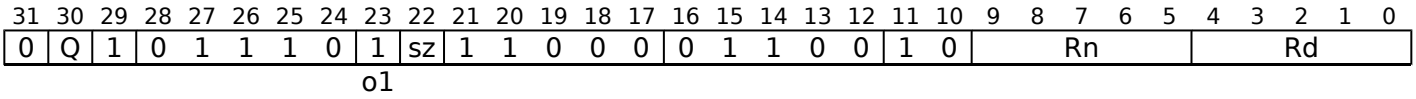
(FEAT_FP16)



FMINNMV <V><d>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
```

Single-precision and double-precision



FMINNMV <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then UNDEFINED;    // .4S only

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, H.

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	x	RESERVED
1	0	4S
1	1	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
V[d, esize] = Reduce(ReduceOp_FMINNUM, operand, esize, FALSE);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMINP (scalar)

Floating-point Minimum of Pair of elements (scalar). This instruction compares two vector elements in the source SIMD&FP register and writes the smallest of the floating-point values as a scalar to the destination SIMD&FP register.

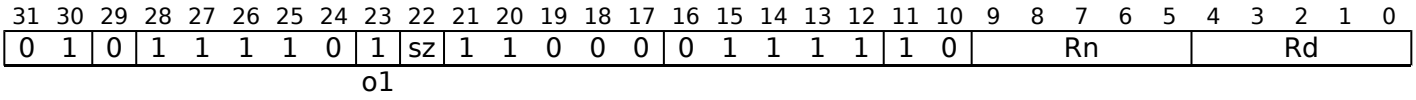
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

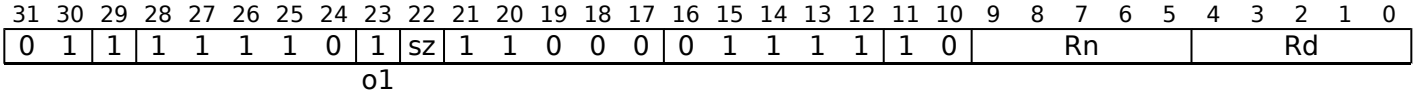
(FEAT_FP16)



FMINP <V><d>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
if sz == '1' then UNDEFINED;
integer datasize = 32;
```

Single-precision and double-precision



FMINP <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize * 2;
```

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	H
1	RESERVED

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2H
1	RESERVED

For the single-precision and double-precision variant: is the source arrangement specifier, encoded in “sz”:

sz	<T>
0	2S
1	2D

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

V[d, esize] = Reduce(ReduceOp_FMIN, operand, esize);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMINP (vector)

Floating-point Minimum Pairwise (vector). This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements from the concatenated vector, writes the smaller of each pair of values into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

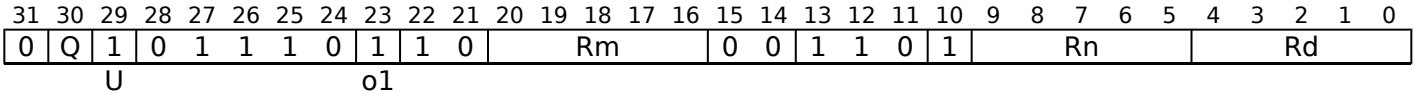
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

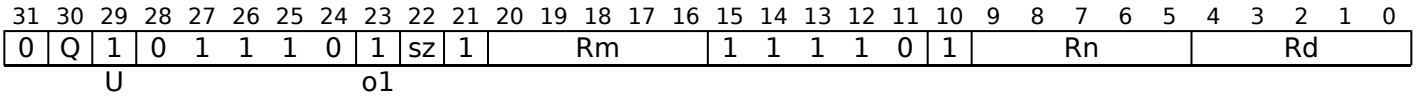
(FEAT_FP16)



FMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Single-precision and double-precision



FMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean pair = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T>	For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if pair then
        element1 = Elem[concat, 2*e, esize];
        element2 = Elem[concat, (2*e)+1, esize];
    else
        element1 = Elem[operand1, e, esize];
        element2 = Elem[operand2, e, esize];

    if minimum then
        Elem[result, e, esize] = FPMin(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = FPMax(element1, element2, FPCR[]);

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMINV

Floating-point Minimum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are floating-point values.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	0	1	1	0	0	0	0	1	1	1	1	1	0	Rn					Rd				
								o1																							

FMINV <V><d>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	1	0	0	0	0	1	1	1	1	1	0	Rn					Rd				
								o1																							

FMINV <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q != '01' then UNDEFINED;

integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

<V> For the half-precision variant: is the destination width specifier, H.

For the single-precision and double-precision variant: is the destination width specifier, encoded in “sz”:

sz	<V>
0	S
1	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	x	RESERVED
1	0	4S
1	1	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

V[d, esize] = Reduce(ReduceOp_FMIN, operand, esize);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMLA (by element)

Floating-point fused Multiply-Add to accumulator (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the results in the vector elements of the destination SIMD&FP register. All the values in this instruction are floating-point values.

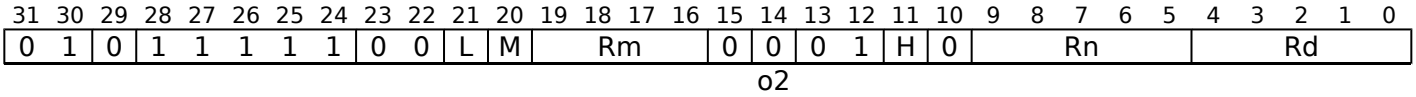
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar, half-precision , Scalar, single-precision and double-precision , Vector, half-precision and Vector, single-precision and double-precision

Scalar, half-precision

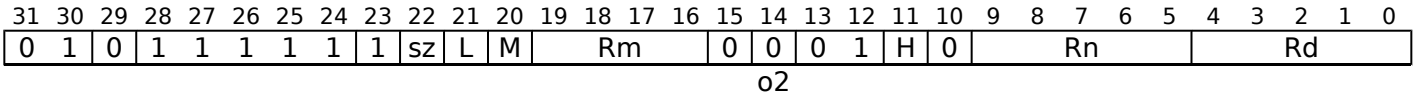
(FEAT_FP16)



FMLA <Hd>, <Hn>, <Vm>.H[<index>]

```
if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');
```

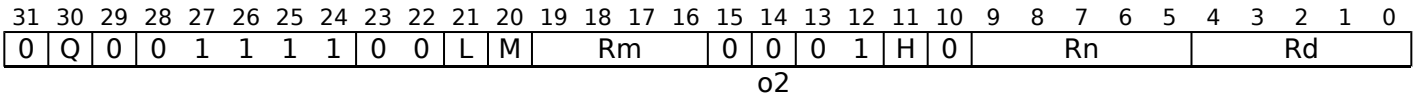
Scalar, single-precision and double-precision



FMLA <V><d>, <V><n>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');
```

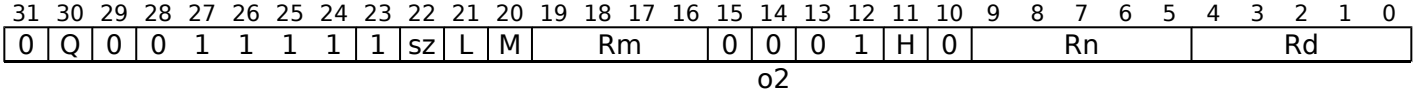
Vector, half-precision
(FEAT_FP16)



FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]

```
if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');
```

Vector, single-precision and double-precision



FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	0	2S
0	1	RESERVED
1	0	4S
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> Is an element size specifier, encoded in “sz”:

sz	<Ts>
0	S
1	D

<index> For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the single-precision and double-precision variant: is the element index, encoded in “sz:L:H”:

sz	L	<index>
0	x	H:L
1	0	H
1	1	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(datasize) operand3 = V[d, datasize];
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, fpcr);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMLA (vector)

Floating-point fused Multiply-Add to accumulator (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, adds the product to the corresponding vector element of the destination SIMD&FP register, and writes the result to the destination SIMD&FP register.

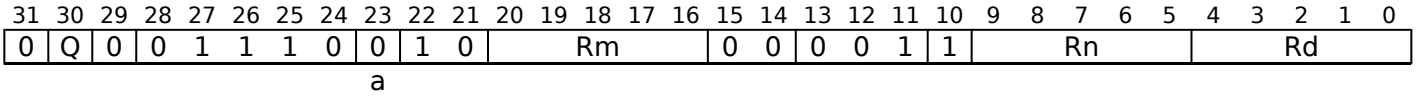
A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

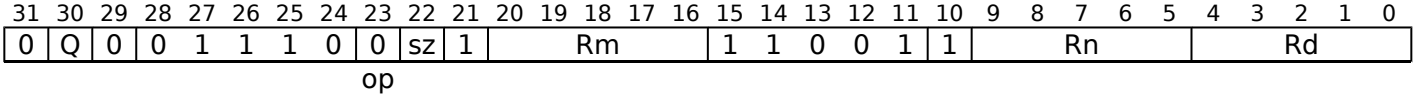
(FEAT_FP16)



FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (a == '1');
```

Single-precision and double-precision



FMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (op == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR[]);

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMLAL, FMLAL2 (by element)

Floating-point fused Multiply-Add Long to accumulator (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

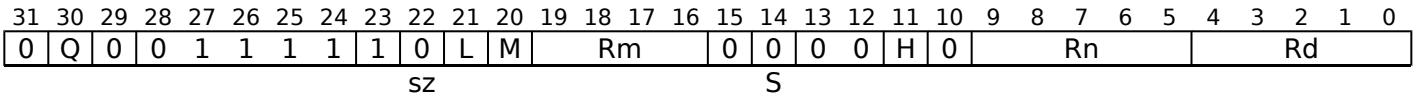
Note

ID_AA64ISAR0_EL1.FHM indicates whether this instruction is supported.

It has encodings from 2 classes: FMLAL and FMLAL2

FMLAL

(FEAT_FHM)

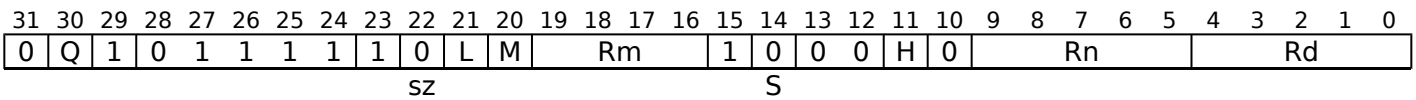


FMLAL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt('0':Rm);    // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
integer index = UInt(H:L:M);

integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 0;
```

FMLAL2

(FEAT_FHM)



FMLAL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt('0':Rm);    // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
integer index = UInt(H:L:M);

integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 1;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	2H
1	4H

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<index> Is the element index, encoded in the "H:L:M" fields.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n, part, datasize DIV 2];
bits(128) operand2 = V[m, 128];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2 = Elem[operand2, index, esize DIV 2];

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize DIV 2];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR[]);
V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMLAL, FMLAL2 (vector)

Floating-point fused Multiply-Add Long to accumulator (vector). This instruction multiplies corresponding half-precision floating-point values in the vectors in the two source SIMD&FP registers, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

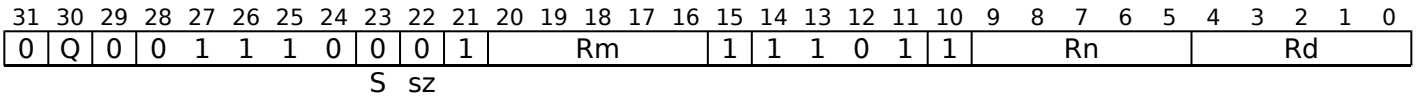
Note

ID_AA64ISAR0_EL1.FHM indicates whether this instruction is supported.

It has encodings from 2 classes: FMLAL and FMLAL2

FMLAL

(FEAT_FHM)

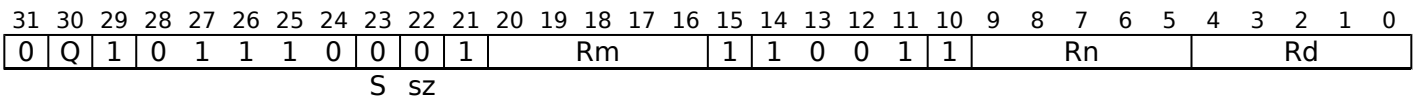


FMLAL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 0;
```

FMLAL2

(FEAT_FHM)



FMLAL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 1;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	2H
1	4H

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n, part, datasize DIV 2];
bits(datasize DIV 2) operand2 = Vpart[m, part, datasize DIV 2];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize DIV 2];
    element2 = Elem[operand2, e, esize DIV 2];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR[]);
V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMLS (by element)

Floating-point fused Multiply-Subtract from accumulator (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and subtracts the results from the vector elements of the destination SIMD&FP register. All the values in this instruction are floating-point values.

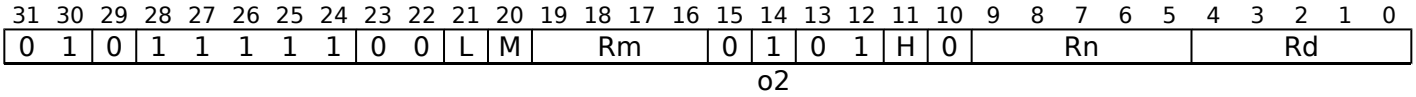
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar, half-precision , Scalar, single-precision and double-precision , Vector, half-precision and Vector, single-precision and double-precision

Scalar, half-precision

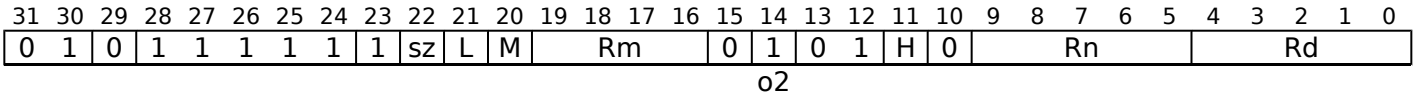
(FEAT_FP16)



FMLS <Hd>, <Hn>, <Vm>.H[<index>]

```
if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');
```

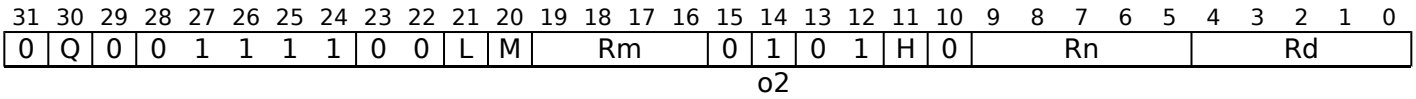
Scalar, single-precision and double-precision



FMLS <V><d>, <V><n>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (o2 == '1');
```

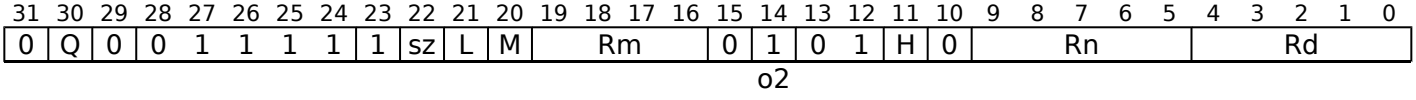
Vector, half-precision
(FEAT_FP16)



FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]

```
if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');
```

Vector, single-precision and double-precision



FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (o2 == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	0	2S
0	1	RESERVED
1	0	4S
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> Is an element size specifier, encoded in “sz”:

sz	<Ts>
0	S
1	D

<index> For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the single-precision and double-precision variant: is the element index, encoded in “sz:L:H”:

sz	L	<index>
0	x	H:L
1	0	H
1	1	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(datasize) operand3 = V[d, datasize];
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, fpcr);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMLS (vector)

Floating-point fused Multiply-Subtract from accumulator (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, negates the product, adds the result to the corresponding vector element of the destination SIMD&FP register, and writes the result to the destination SIMD&FP register.

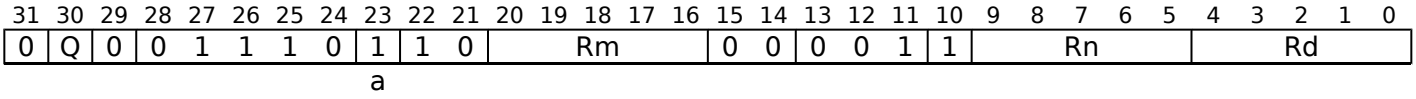
A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

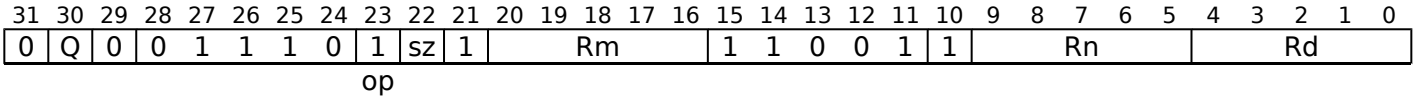
(FEAT_FP16)



FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (a == '1');
```

Single-precision and double-precision



FMLS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (op == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(Elem[operand3, e, esize], element1, element2, FPCR[]);

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMLSL, FMLSL2 (by element)

Floating-point fused Multiply-Subtract Long from accumulator (by element). This instruction multiplies the negated vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

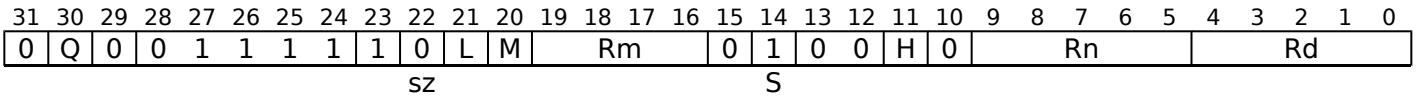
Note

ID_AA64ISAR0_EL1.FHM indicates whether this instruction is supported.

It has encodings from 2 classes: FMLSL and FMLSL2

FMLSL

(FEAT_FHM)

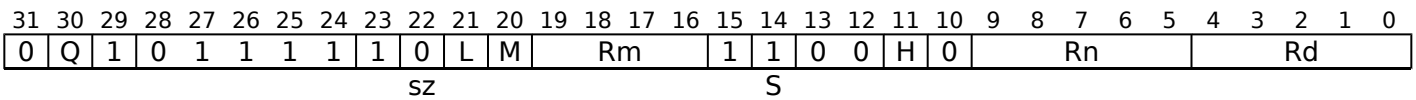


FMLSL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt('0':Rm);    // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
integer index = UInt(H:L:M);

integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 0;
```

FMLSL2

(FEAT_FHM)



FMLSL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt('0':Rm);    // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
integer index = UInt(H:L:M);

integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 1;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	2H
1	4H

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<index> Is the element index, encoded in the "H:L:M" fields.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n, part, datasize DIV 2];
bits(128) operand2 = V[m, 128];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2 = Elem[operand2, index, esize DIV 2];

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize DIV 2];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR[]);
V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMLSL, FMLSL2 (vector)

Floating-point fused Multiply-Subtract Long from accumulator (vector). This instruction negates the values in the vector of one SIMD&FP register, multiplies these with the corresponding values in another vector, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

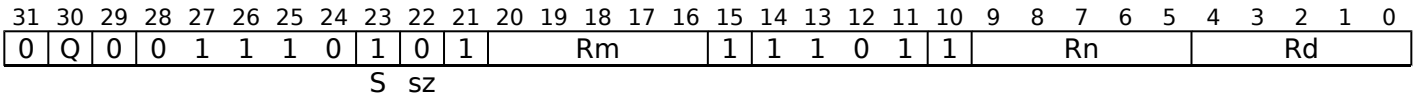
Note

ID_AA64ISAR0_EL1.FHM indicates whether this instruction is supported.

It has encodings from 2 classes: FMLSL and FMLSL2

FMLSL

(FEAT_FHM)

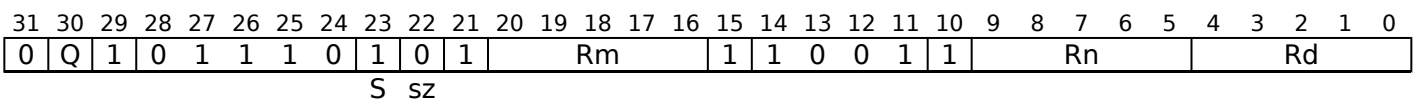


FMLSL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 0;
```

FMLSL2

(FEAT_FHM)



FMLSL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (S == '1');
integer part = 1;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	2H
1	4H

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

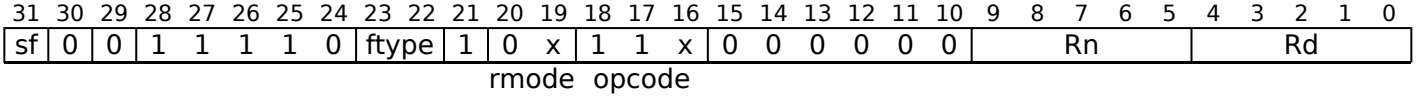
```
CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n, part, datasize DIV 2];
bits(datasize DIV 2) operand2 = Vpart[m, part, datasize DIV 2];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize DIV 2];
    element2 = Elem[operand2, e, esize DIV 2];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, FPCR[]);
V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMOV (general)

Floating-point Move to or from general-purpose register without conversion. This instruction transfers the contents of a SIMD&FP register to a general-purpose register, or the contents of a general-purpose register to a SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision to 32-bit (sf == 0 && ftype == 11 && rmode == 00 && opcode == 110)
(FEAT_FP16)

```
FMOV <Wd>, <Hn>
```

Half-precision to 64-bit (sf == 1 && ftype == 11 && rmode == 00 && opcode == 110)
(FEAT_FP16)

```
FMOV <Xd>, <Hn>
```

32-bit to half-precision (sf == 0 && ftype == 11 && rmode == 00 && opcode == 111)
(FEAT_FP16)

```
FMOV <Hd>, <Wn>
```

32-bit to single-precision (sf == 0 && ftype == 00 && rmode == 00 && opcode == 111)

```
FMOV <Sd>, <Wn>
```

Single-precision to 32-bit (sf == 0 && ftype == 00 && rmode == 00 && opcode == 110)

```
FMOV <Wd>, <Sn>
```

64-bit to half-precision (sf == 1 && ftype == 11 && rmode == 00 && opcode == 111)
(FEAT_FP16)

```
FMOV <Hd>, <Xn>
```

64-bit to double-precision (sf == 1 && ftype == 01 && rmode == 00 && opcode == 111)

```
FMOV <Dd>, <Xn>
```

64-bit to top half of 128-bit (sf == 1 && ftype == 10 && rmode == 01 && opcode == 111)

```
FMOV <Vd>.D[1], <Xn>
```

Double-precision to 64-bit (sf == 1 && ftype == 01 && rmode == 00 && opcode == 110)

```
FMOV <Xd>, <Dn>
```

Top half of 128-bit to 64-bit (sf == 1 && ftype == 10 && rmode == 01 && opcode == 110)

```
FMOV <Xd>, <Vn>.D[1]
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPConvOp op;
FPRounding rounding;
boolean unsigned;
integer part;

case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        if opcode<2:1>:rmode != '11 01' then UNDEFINED;
        fltsize = 128;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

case opcode<2:1>:rmode of
    when '00 xx'    // FCVT[NPMZ][US]
        rounding = FPDecodeRounding(rmode);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '01 00'    // [US]CVTF
        rounding = FPRoundingMode(FPCR[]);
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_ItoF;
    when '10 00'    // FCVTA[US]
        rounding = FPRounding_TIEAWAY;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI;
    when '11 00'    // FMOV
        if fltsize != 16 && fltsize != intsize then UNDEFINED;
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 0;
    when '11 01'    // FMOV D[1]
        if intsize != 64 || fltsize != 128 then UNDEFINED;
        op = if opcode<0> == '1' then FPConvOp_MOV_ItoF else FPConvOp_MOV_FtoI;
        part = 1;
        fltsize = 64;    // size of D[1] is 64
    when '11 11'    // FJCVTZS
        if !HaveFJCVTZSExt() then UNDEFINED;
        rounding = FPRounding_ZERO;
        unsigned = (opcode<0> == '1');
        op = FPConvOp_CVT_FtoI_JS;
    otherwise
        UNDEFINED;
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<Dn> Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
if op == FPConvOp_CVT_FtoI_JS then
    CheckFPAdvSIMDEnabled64();
else
    CheckFPEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
integer fsize = if op == FPConvOp_CVT_ItoF && merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;

case op of
    when FPConvOp_CVT_FtoI
        fltval = V[n, fsize];
        intval = FPToFixed(fltval, 0, unsigned, fpcr, rounding, intsize);
        X[d, intsize] = intval;
    when FPConvOp_CVT_ItoF
        intval = X[n, intsize];
        fltval = if merge then V[d, fsize] else Zeros(fsize);
        Elem[fltval, 0, fltsize] = FixedToFP(intval, 0, unsigned, fpcr, rounding, fltsize);
        V[d, fsize] = fltval;
    when FPConvOp_MOV_FtoI
        fltval = Vpart[n, part, fsize];
        intval = ZeroExtend(fltval, intsize);
        X[d, intsize] = intval;
    when FPConvOp_MOV_ItoF
        intval = X[n, intsize];
        fltval = intval<fsize-1:0>;
        Vpart[d, part, fsize] = fltval;
    when FPConvOp_CVT_FtoI_JS
        bit z;
        fltval = V[n, fsize];
        (intval, z) = FPToFixedJS(fltval, fpcr, TRUE, intsize);
        PSTATE.<N,Z,C,V> = '0':z:'00';
        X[d, intsize] = intval;
```

Operational information

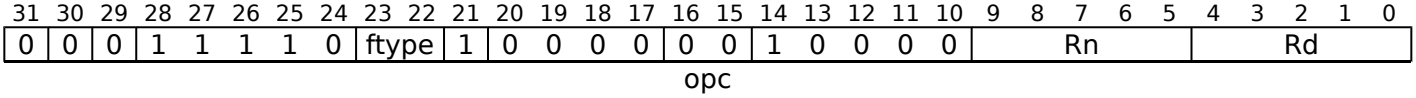
If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMOV (register)

Floating-point Move register without conversion. This instruction copies the floating-point value in the SIMD&FP source register to the SIMD&FP destination register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)
(FEAT_FP16)

```
FMOV <Hd>, <Hn>
```

Single-precision (ftype == 00)

```
FMOV <Sd>, <Sn>
```

Double-precision (ftype == 01)

```
FMOV <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

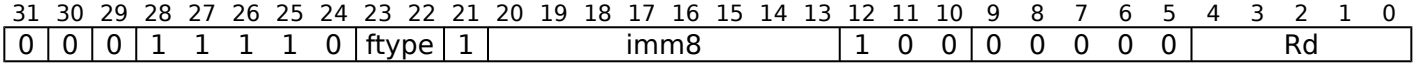
Operation

```
CheckFPEnabled64();

bits(128) result = 0<127:0>;

bits(esize) operand = V[n, esize];

Elem[result, 0, esize] = operand;
V[d, 128] = result;
```

FMOV (scalar, immediate)

Floating-point move immediate (scalar). This instruction copies a floating-point immediate constant into the SIMD&FP destination register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)
(FEAT_FP16)

```
FMOV <Hd>, #<imm>
```

Single-precision (ftype == 00)

```
FMOV <Sd>, #<imm>
```

Double-precision (ftype == 01)

```
FMOV <Dd>, #<imm>
```

```
integer d = UInt(Rd);

integer datasize;
case ftype of
    when '00' datasize = 32;
    when '01' datasize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            datasize = 16;
        else
            UNDEFINED;

bits(datasize) imm = VFPExpandImm(imm8, datasize);
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<imm>	Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in the "imm8" field. For details of the range of constants available and the encoding of <imm>, see *Modified immediate constants in A64 floating-point instructions*.

Operation

```
CheckFPEnabled64();

V[d, datasize] = imm;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMOV (vector, immediate)

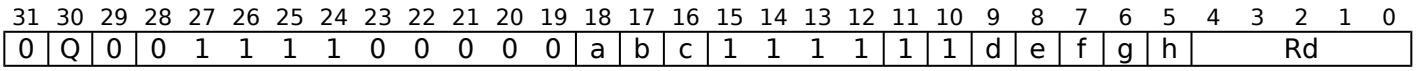
Floating-point move immediate (vector). This instruction copies an immediate floating-point constant into every element of the SIMD&FP destination register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

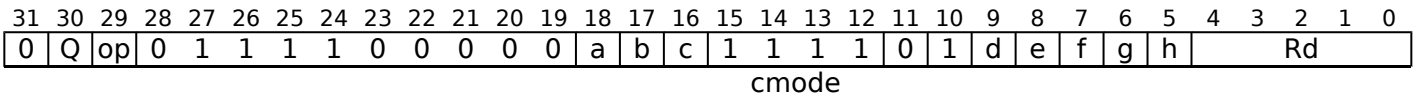
(FEAT_FP16)



FMOV <Vd>.<T>, #<imm>

```
if !HaveFP16Ext() then UNDEFINED;

integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;

imm8 = a:b:c:d:e:f:g:h;
imm16 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>, 2):imm8<5:0>:Zeros(6);

imm = Replicate(imm16, datasize DIV 16);
```

Single-precision and double-precision



Single-precision (op == 0)

FMOV <Vd>.<T>, #<imm>

Double-precision (Q == 1 && op == 1)

FMOV <Vd>.2D, #<imm>

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

if cmode:op == '11111' then
    // FMOV Dn,#imm is in main FP instruction set
    if Q == '0' then UNDEFINED;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<T>	For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	2S
1	4S

<imm> Is a signed floating-point constant with 3-bit exponent and normalized 4 bits of precision, encoded in "a:b:c:d:e:f:g:h". For details of the range of constants available and the encoding of <imm>, see *Modified immediate constants in A64 floating-point instructions*.

Operation

```
CheckFPAdvSIMDEnabled64();

V[rd, datasize] = imm;
```

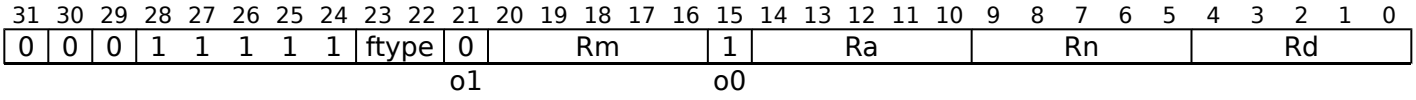
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMSUB

Floating-point Fused Multiply-Subtract (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, negates the product, adds that to the value of the third SIMD&FP source register, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)

(FEAT_FP16)

```
FMSUB <Hd>, <Hn>, <Hm>, <Ha>
```

Single-precision (ftype == 00)

```
FMSUB <Sd>, <Sn>, <Sm>, <Sa>
```

Double-precision (ftype == 01)

```
FMSUB <Dd>, <Dn>, <Dm>, <Da>
```

```
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Da>	Is the 64-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

<Ha> Is the 16-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

<Sa> Is the 32-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

Operation

```
CheckFPEnabled64();

bits(esize) operanda = V[a, esize];
bits(esize) operand1 = V[n, esize];
bits(esize) operand2 = V[m, esize];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[a, 128] else Zeros(128);

operand1 = FPNeg(operand1);
Elem[result, 0, esize] = FPMulAdd(operanda, operand1, operand2, fpcr);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMUL (by element)

Floating-point Multiply (by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are floating-point values.

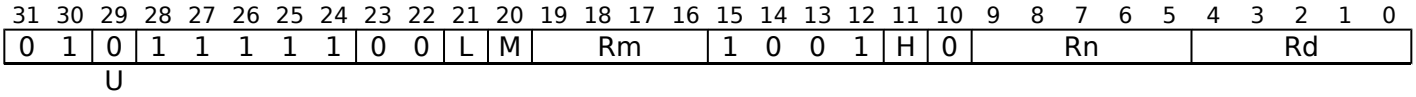
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar, half-precision , Scalar, single-precision and double-precision , Vector, half-precision and Vector, single-precision and double-precision

Scalar, half-precision

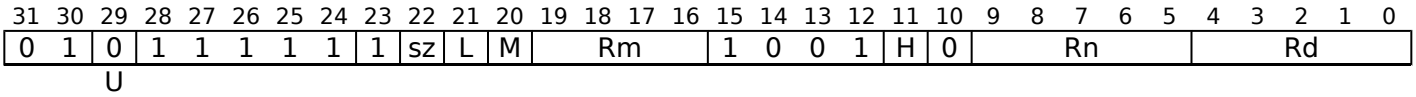
(FEAT_FP16)



FMUL <Hd>, <Hn>, <Vm>.H[<index>]

```
if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');
```

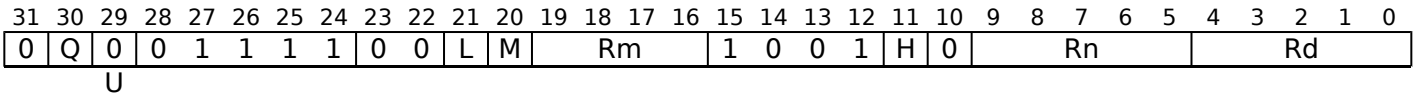
Scalar, single-precision and double-precision



FMUL <V><d>, <V><n>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');
```

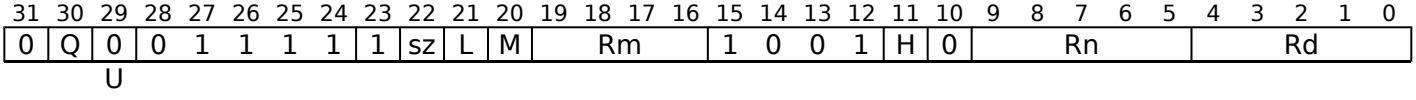
Vector, half-precision
(FEAT_FP16)



FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]

```
if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');
```

Vector, single-precision and double-precision



FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	0	2S
0	1	RESERVED
1	0	4S
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> Is an element size specifier, encoded in “sz”:

sz	<Ts>
0	S
1	D

<index> For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the single-precision and double-precision variant: is the element index, encoded in “sz:L:H”:

sz	L	<index>
0	x	H:L
1	0	H
1	1	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if mulx_op then
        Elem[result, e, esize] = FPMulX(element1, element2, fpcr);
    else
        Elem[result, e, esize] = FPMul(element1, element2, fpcr);

V[d, 128] = result;
```

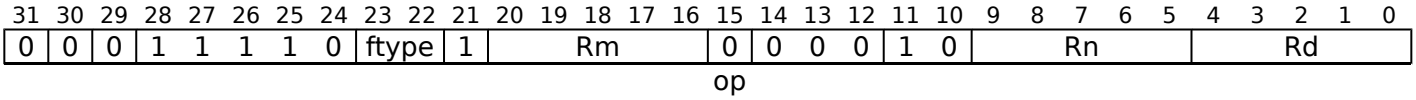
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMUL (scalar)

Floating-point Multiply (scalar). This instruction multiplies the floating-point values of the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)

(FEAT_FP16)

```
FMUL <Hd>, <Hn>, <Hm>
```

Single-precision (ftype == 00)

```
FMUL <Sd>, <Sn>, <Sm>
```

Double-precision (ftype == 01)

```
FMUL <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPEnabled64();
bits(esize) operand1 = V[n, esize];
bits(esize) operand2 = V[m, esize];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

bits(esize) product = FPMul(operand1, operand2, fpcr);
Elem[result, 0, esize] = product;

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMUL (vector)

Floating-point Multiply (vector). This instruction multiplies corresponding floating-point values in the vectors in the two source SIMD&FP registers, places the result in a vector, and writes the vector to the destination SIMD&FP register.

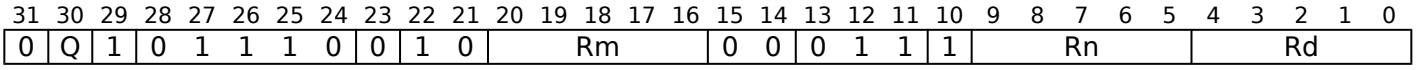
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

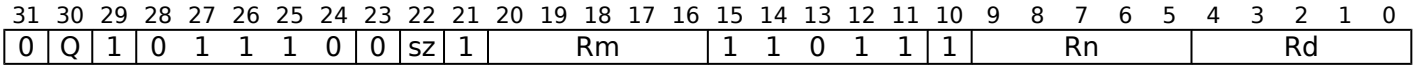
(FEAT_FP16)



FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Single-precision and double-precision



FMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMul(element1, element2, FPCR[]);

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMULX

Floating-point Multiply extended. This instruction multiplies corresponding floating-point values in the vectors of the two source SIMD&FP registers, places the resulting floating-point values in a vector, and writes the vector to the destination SIMD&FP register.

If one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

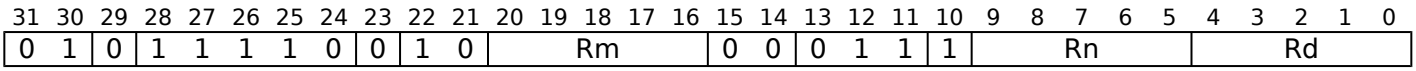
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

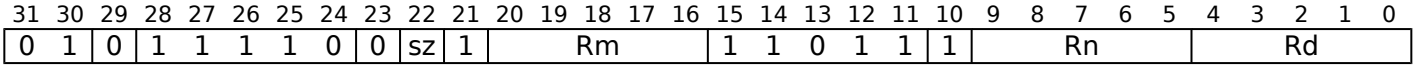
(FEAT_FP16)



FMULX <Hd>, <Hn>, <Hm>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
```

Scalar single-precision and double-precision

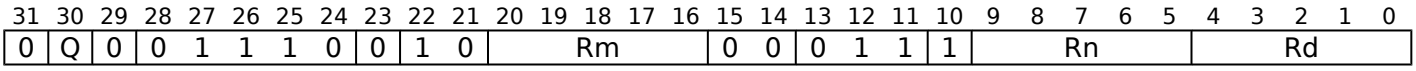


FMULX <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

Vector half precision

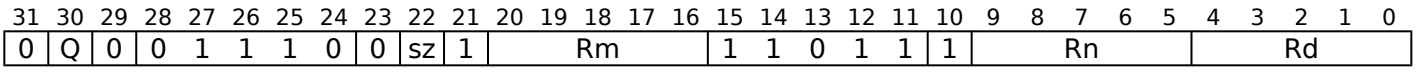
(FEAT_FP16)



FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Vector single-precision and double-precision



FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
if elements == 1 then
    CheckFPEnabled64();
else
    CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMulX(element1, element2, fpcr);
V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMULX (by element)

Floating-point Multiply extended (by element). This instruction multiplies the floating-point values in the vector elements in the first source SIMD&FP register by the specified floating-point value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If one value is zero and the other value is infinite, the result is 2.0. In this case, the result is negative if only one of the values is negative, otherwise the result is positive.

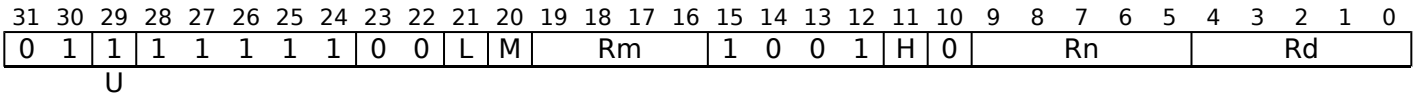
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar, half-precision , Scalar, single-precision and double-precision , Vector, half-precision and Vector, single-precision and double-precision

Scalar, half-precision

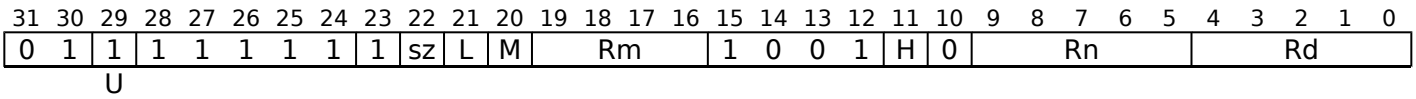
(FEAT_FP16)



FMULX <Hd>, <Hn>, <Vm>.H[<index>]

```
if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');
```

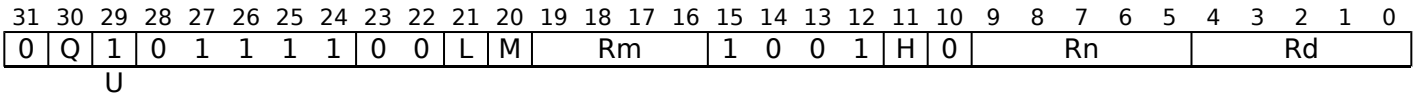
Scalar, single-precision and double-precision



FMULX <V><d>, <V><n>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean mulx_op = (U == '1');
```

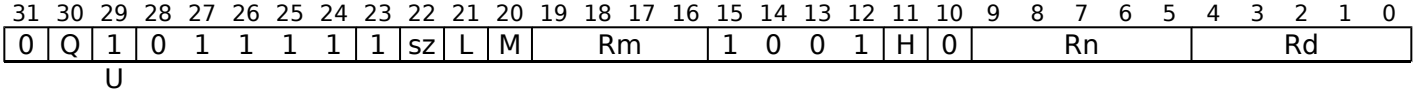
Vector, half-precision
(FEAT_FP16)



FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.H[<index>]

```
if !HaveFP16Ext() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer index = UInt(H:L:M);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');
```

Vector, single-precision and double-precision



FMULX <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi = M;
case sz:L of
    when '0x' index = UInt(H:L);
    when '10' index = UInt(H);
    when '11' UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean mulx_op = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “Q:sz”:

Q	sz	<T>
0	0	2S
0	1	RESERVED
1	0	4S
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> For the half-precision variant: is the name of the second SIMD&FP source register, in the range V0 to V15, encoded in the "Rm" field.

For the single-precision and double-precision variant: is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<Ts> Is an element size specifier, encoded in “sz”:

sz	<Ts>
0	S
1	D

<index> For the half-precision variant: is the element index, in the range 0 to 7, encoded in the "H:L:M" fields.

For the single-precision and double-precision variant: is the element index, encoded in “sz:L:H”:

sz	L	<index>
0	x	H:L
1	0	H
1	1	RESERVED

Operation

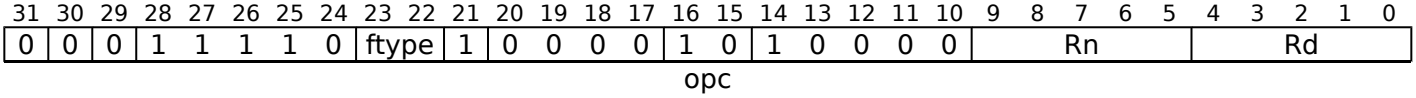
```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(esize) element1;
bits(esize) element2 = Elem[operand2, index, esize];
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    if mulx_op then
        Elem[result, e, esize] = FPMulX(element1, element2, fpcr);
    else
        Elem[result, e, esize] = FPMul(element1, element2, fpcr);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FNEG (scalar)

Floating-point Negate (scalar). This instruction negates the value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)
(FEAT_FP16)

```
FNEG <Hd>, <Hn>
```

Single-precision (ftype == 00)

```
FNEG <Sd>, <Sn>
```

Double-precision (ftype == 01)

```
FNEG <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else 0<127:0>;

bits(esize) operand = V[n, esize];

Elem[result, 0, esize] = FPNeg(operand);
V[d, 128] = result;
```

FNEG (vector)

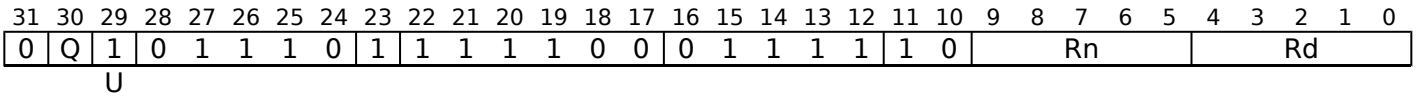
Floating-point Negate (vector). This instruction negates the value of each vector element in the source SIMD&FP register, writes the result to a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

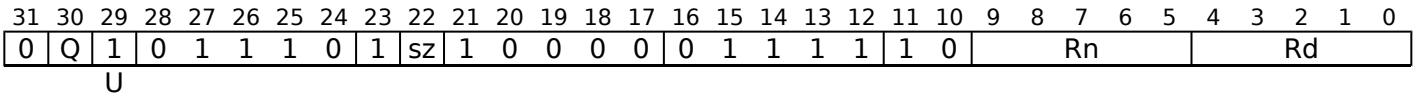
(FEAT_FP16)



FNEG <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Single-precision and double-precision



FNEG <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    if neg then
        element = FPNeg(element);
    else
        element = FPAbs(element);
    Elem[result, e, esize] = element;

V[d, datasize] = result;
```

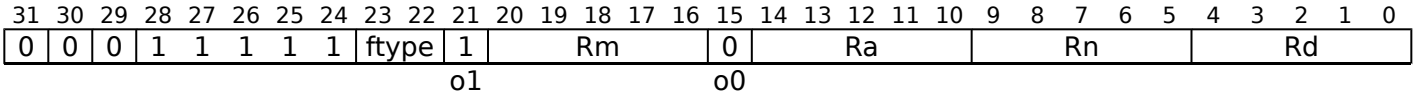
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FNMADD

Floating-point Negated fused Multiply-Add (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, negates the product, subtracts the value of the third SIMD&FP source register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)

(FEAT_FP16)

```
FNMADD <Hd>, <Hn>, <Hm>, <Ha>
```

Single-precision (ftype == 00)

```
FNMADD <Sd>, <Sn>, <Sm>, <Sa>
```

Double-precision (ftype == 01)

```
FNMADD <Dd>, <Dn>, <Dm>, <Da>
```

```
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Da>	Is the 64-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Ha>	Is the 16-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

<Sa> Is the 32-bit name of the third SIMD&FP source register holding the addend, encoded in the "Ra" field.

Operation

```
CheckFPEnabled64();

bits(esize) operanda = V[a, esize];
bits(esize) operand1 = V[n, esize];
bits(esize) operand2 = V[m, esize];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[a, 128] else Zeros(128);

operanda = FPNeg(operanda);
operand1 = FPNeg(operand1);
Elem[result, 0, esize] = FPMulAdd(operanda, operand1, operand2, fpcr);

V[d, 128] = result;
```

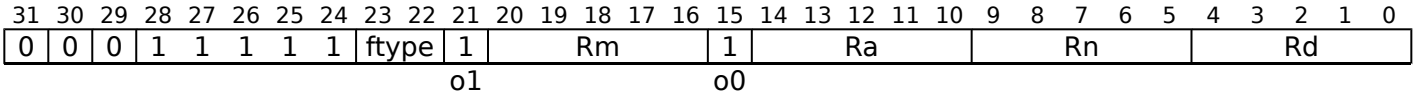
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FNMSUB

Floating-point Negated fused Multiply-Subtract (scalar). This instruction multiplies the values of the first two SIMD&FP source registers, subtracts the value of the third SIMD&FP source register, and writes the result to the destination SIMD&FP register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)

(FEAT_FP16)

```
FNMSUB <Hd>, <Hn>, <Hm>, <Ha>
```

Single-precision (ftype == 00)

```
FNMSUB <Sd>, <Sn>, <Sm>, <Sa>
```

Double-precision (ftype == 01)

```
FNMSUB <Dd>, <Dn>, <Dm>, <Da>
```

```
integer d = UInt(Rd);
integer a = UInt(Ra);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.
<Da>	Is the 64-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

<Ha> Is the 16-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Sn> Is the 32-bit name of the first SIMD&FP source register holding the multiplicand, encoded in the "Rn" field.

<Sm> Is the 32-bit name of the second SIMD&FP source register holding the multiplier, encoded in the "Rm" field.

<Sa> Is the 32-bit name of the third SIMD&FP source register holding the minuend, encoded in the "Ra" field.

Operation

```
CheckFPEnabled64();

bits(esize) operanda = V[a, esize];
bits(esize) operand1 = V[n, esize];
bits(esize) operand2 = V[m, esize];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[a, 128] else Zeros(128);

operanda = FPNeg(operanda);
Elem[result, 0, esize] = FPMulAdd(operanda, operand1, operand2, fpcr);

V[d, 128] = result;
```

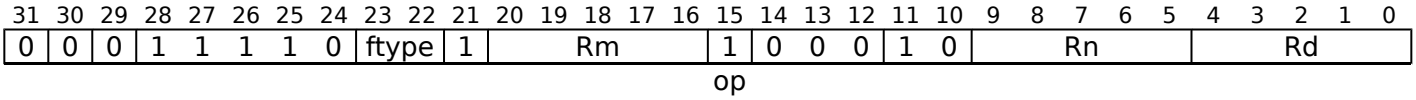
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FNMUL (scalar)

Floating-point Multiply-Negate (scalar). This instruction multiplies the floating-point values of the two source SIMD&FP registers, and writes the negation of the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)

(FEAT_FP16)

```
FNMUL <Hd>, <Hn>, <Hm>
```

Single-precision (ftype == 00)

```
FNMUL <Sd>, <Sn>, <Sm>
```

Double-precision (ftype == 01)

```
FNMUL <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPEnabled64();
bits(esize) operand1 = V[n, esize];
bits(esize) operand2 = V[m, esize];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

bits(esize) product = FPMul(operand1, operand2, fpcr);
product = FPNeg(product);
Elem[result, 0, esize] = product;

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FRECPE

Floating-point Reciprocal Estimate. This instruction finds an approximate reciprocal estimate for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

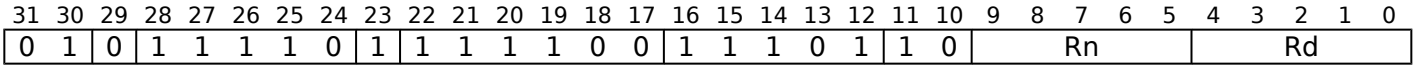
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

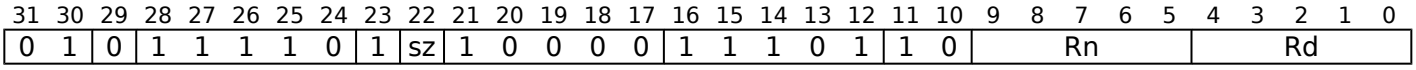
(FEAT_FP16)



FRECPE <Hd>, <Hn>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
```

Scalar single-precision and double-precision

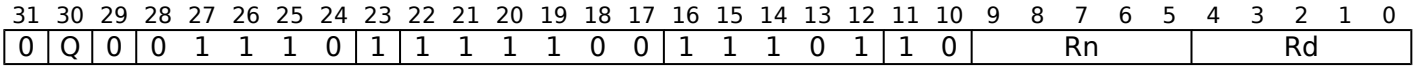


FRECPE <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

Vector half precision

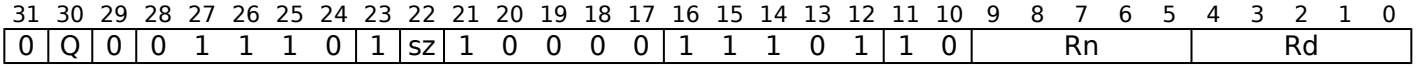
(FEAT_FP16)



FRECPE <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Vector single-precision and double-precision



FRECPE <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
if elements == 1 then
    CheckFPEnabled64();
else
    CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRecipEstimate(element, FPCR[]);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FRECPS

Floating-point Reciprocal Step. This instruction multiplies the corresponding floating-point values in the vectors of the two source SIMD&FP registers, subtracts each of the products from 2.0, places the resulting floating-point values in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	1	0	Rm					0	0	1	1	1	1	Rn					Rd				

FRECPS <Hd>, <Hn>, <Hm>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	0	sz	1	Rm					1	1	1	1	1	1	Rn					Rd				

FRECPS <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

Vector half precision

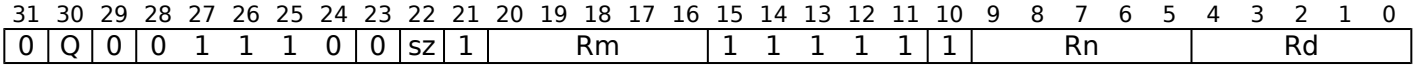
(FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	0	1	0	Rm					0	0	1	1	1	1	Rn					Rd				

FRECPS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Vector single-precision and double-precision



FRECPS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
if elements == 1 then
    CheckFPEnabled64();
else
    CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPRecipStepFused(element1, element2);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FRECPX

Floating-point Reciprocal exponent (scalar). This instruction finds an approximate reciprocal exponent for the source SIMD&FP register and writes the result to the destination SIMD&FP register.

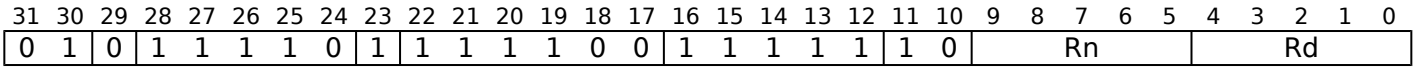
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

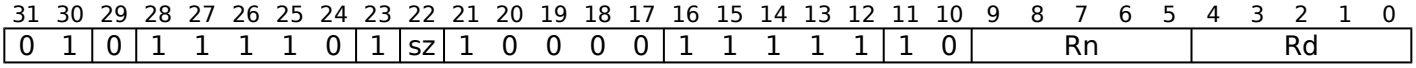
(FEAT_FP16)



FRECPX <Hd>, <Hn>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
```

Single-precision and double-precision



FRECPX <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEnabled64();
bits(esize) operand = V[n, esize];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

Elem[result, 0, esize] = FPRecpX(operand, fpcr);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FRINT32X (scalar)

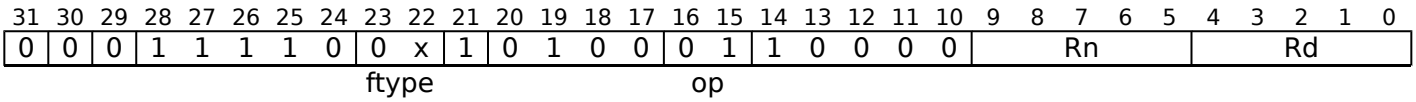
Floating-point Round to 32-bit Integer, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 32-bit integer size using the rounding mode that is determined by the *FPCR*, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns {for the corresponding result value} the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point
(FEAT_FRINTTS)



Single-precision (ftype == 00)

```
FRINT32X <Sd>, <Sn>
```

Double-precision (ftype == 01)

```
FRINT32X <Dd>, <Dn>
```

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '1x' UNDEFINED;

FPRounding rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);
bits(esize) operand = V[n, esize];

Elem[result, 0, esize] = FPRoundIntN(operand, fpcr, rounding, 32);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FRINT32X (vector)

Floating-point Round to 32-bit Integer, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 32-bit integer size using the rounding mode that is determined by the *FPCR*, and writes the result to the SIMD&FP destination register.

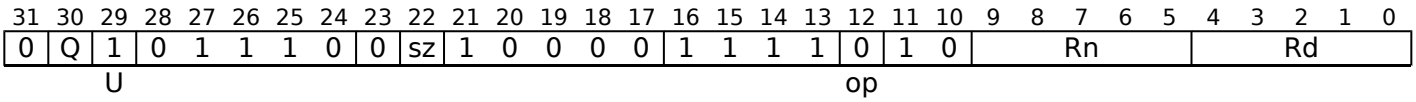
A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision

(FEAT_FRINTTS)



FRINT32X <Vd>.<T>, <Vn>.<T>

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then 32 else 64;
FPRounding rounding = if U == '0' then FPRounding_ZERO else FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundIntN(element, FPCR[], rounding, intsize);

V[d, datasize] = result;
```

FRINT32Z (scalar)

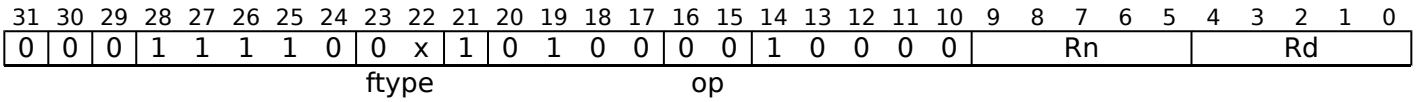
Floating-point Round to 32-bit Integer toward Zero (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 32-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the {corresponding} input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns {for the corresponding result value} the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point
(FEAT_FRINTTS)



Single-precision (ftype == 00)

```
FRINT32Z <Sd>, <Sn>
```

Double-precision (ftype == 01)

```
FRINT32Z <Dd>, <Dn>
```

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '1x' UNDEFINED;
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);
bits(esize) operand = V[n, esize];

Elem[result, 0, esize] = FPRoundIntN(operand, fpcr, FPRounding_ZERO, 32);

V[d, 128] = result;
```

FRINT32Z (vector)

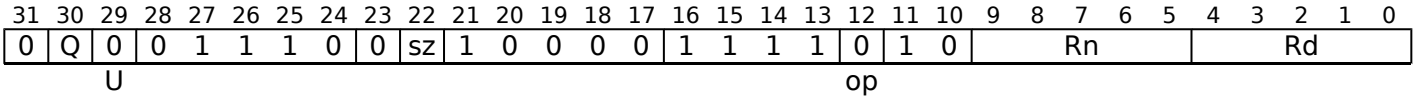
Floating-point Round to 32-bit Integer toward Zero (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 32-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision
(FEAT_FRINTTS)



FRINT32Z <Vd>.<T>, <Vn>.<T>

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then 32 else 64;
FPRounding rounding = if U == '0' then FPRounding_ZERO else FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundIntN(element, FPCR[], rounding, intsize);

V[d, datasize] = result;
```

FRINT64X (scalar)

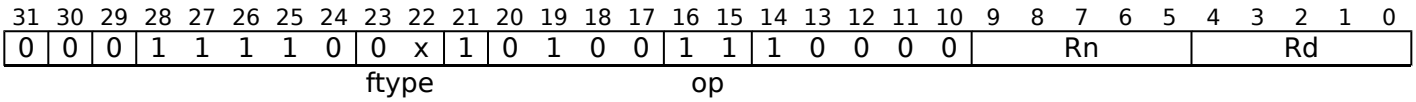
Floating-point Round to 64-bit Integer, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 64-bit integer size using the rounding mode that is determined by the *FPCR*, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns {for the corresponding result value} the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point
(FEAT_FRINTTS)



Single-precision (ftype == 00)

```
FRINT64X <Sd>, <Sn>
```

Double-precision (ftype == 01)

```
FRINT64X <Dd>, <Dn>
```

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '1x' UNDEFINED;

FPRounding rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);
bits(esize) operand = V[n, esize];

Elem[result, 0, esize] = FPRoundIntN(operand, fpcr, rounding, 64);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FRINT64X (vector)

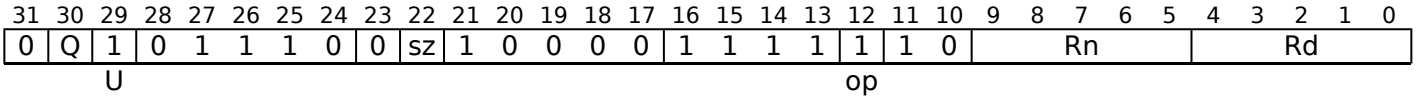
Floating-point Round to 64-bit Integer, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 64-bit integer size using the rounding mode that is determined by the *FPCR*, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision
(FEAT_FRINTTS)



FRINT64X <Vd>.<T>, <Vn>.<T>

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then 32 else 64;
FPRounding rounding = if U == '0' then FPRounding_ZERO else FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundIntN(element, FPCR[], rounding, intsize);

V[d, datasize] = result;
```

FRINT64Z (scalar)

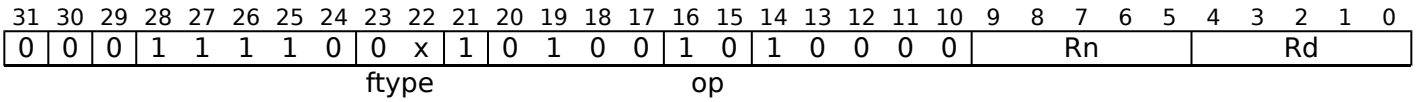
Floating-point Round to 64-bit Integer toward Zero (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value that fits into a 64-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When the result value is not numerically equal to the {corresponding} input value, an Inexact exception is raised. When the input is infinite, NaN or out-of-range, the instruction returns {for the corresponding result value} the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Floating-point
(FEAT_FRINTTS)



Single-precision (ftype == 00)

```
FRINT64Z <Sd>, <Sn>
```

Double-precision (ftype == 01)

```
FRINT64Z <Dd>, <Dn>
```

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '1x' UNDEFINED;
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);
bits(esize) operand = V[n, esize];

Elem[result, 0, esize] = FPRoundIntN(operand, fpcr, FPRounding_ZERO, 64);

V[d, 128] = result;
```

FRINT64Z (vector)

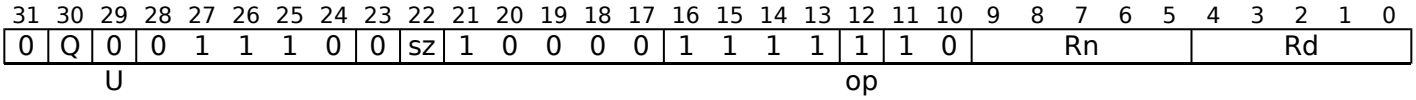
Floating-point Round to 64-bit Integer toward Zero (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values that fit into a 64-bit integer size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input returns a zero result with the same sign. When one of the result values is not numerically equal to the corresponding input value, an Inexact exception is raised. When an input is infinite, NaN or out-of-range, the instruction returns for the corresponding result value the most negative integer representable in the destination size, and an Invalid Operation floating-point exception is raised.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector single-precision and double-precision
(FEAT_FRINTTS)



FRINT64Z <Vd>.<T>, <Vn>.<T>

```
if !HaveFrintExt() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer intsize = if op == '0' then 32 else 64;
FPRounding rounding = if U == '0' then FPRounding_ZERO else FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundIntN(element, FPCR[], rounding, intsize);

V[d, datasize] = result;
```

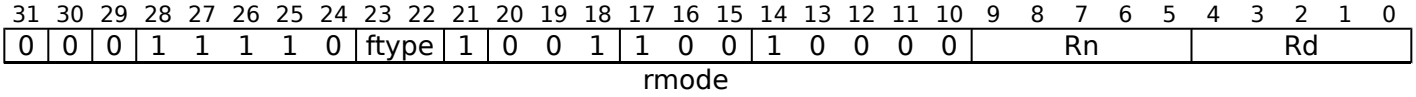
FRINTA (scalar)

Floating-point Round to Integral, to nearest with ties to Away (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round to Nearest with Ties to Away rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)

(FEAT_FP16)

```
FRINTA <Hd>, <Hn>
```

Single-precision (ftype == 00)

```
FRINTA <Sd>, <Sn>
```

Double-precision (ftype == 01)

```
FRINTA <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);
bits(esize) operand = V[n, esize];

Elem[result, 0, esize] = FPRoundInt(operand, fpcr, FPRounding_TIEAWAY, FALSE);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FRINTA (vector)

Floating-point Round to Integral, to nearest with ties to Away (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round to Nearest with Ties to Away rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

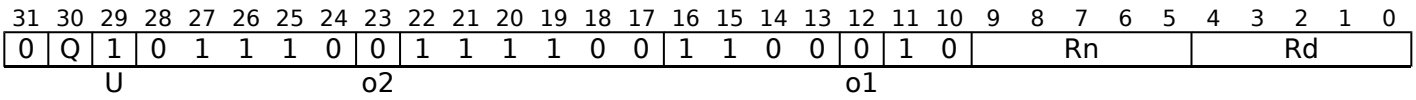
A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

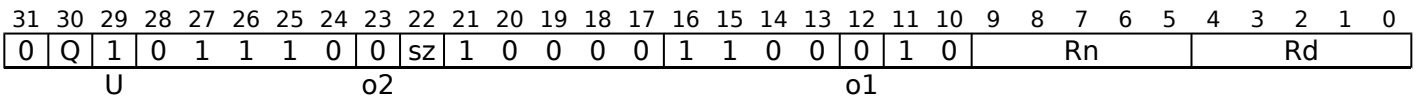
(FEAT_FP16)



FRINTA <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR[]);
```

Single-precision and double-precision



FRINTA <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

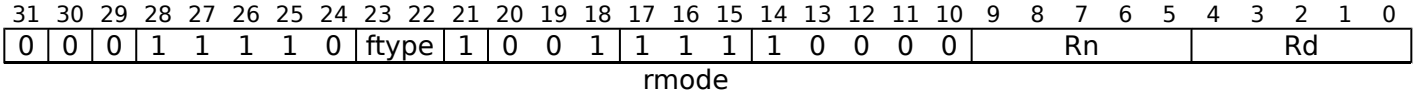
FRINTI (scalar)

Floating-point Round to Integral, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the rounding mode that is determined by the *FPCR*, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)

(FEAT_FP16)

```
FRINTI <Hd>, <Hn>
```

Single-precision (ftype == 00)

```
FRINTI <Sd>, <Sn>
```

Double-precision (ftype == 01)

```
FRINTI <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;

FPRounding rounding;
rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);
bits(esize) operand = V[n, esize];

Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FRINTI (vector)

Floating-point Round to Integral, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the rounding mode that is determined by the *FPCR*, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

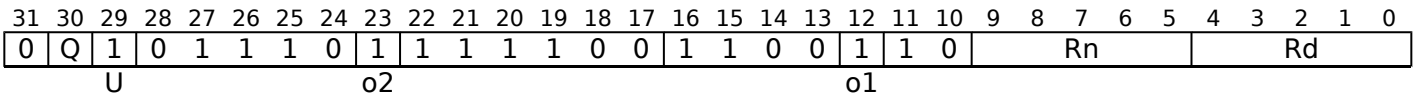
A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

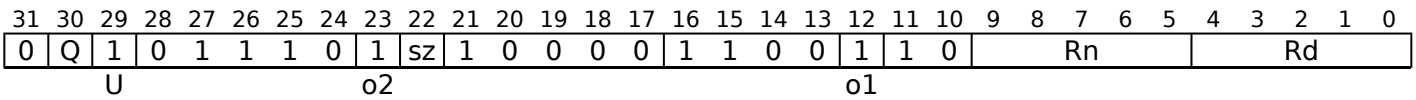
(FEAT_FP16)



FRINTI <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR[]);
```

Single-precision and double-precision



FRINTI <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

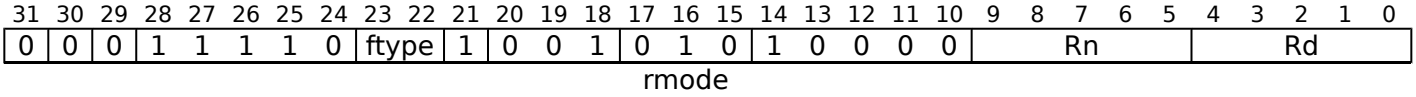
FRINTM (scalar)

Floating-point Round to Integral, toward Minus infinity (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)

(FEAT_FP16)

```
FRINTM <Hd>, <Hn>
```

Single-precision (ftype == 00)

```
FRINTM <Sd>, <Sn>
```

Double-precision (ftype == 01)

```
FRINTM <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;

FPRounding rounding;
rounding = FPDecodeRounding('10');
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);
bits(esize) operand = V[n, esize];

Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FRINTM (vector)

Floating-point Round to Integral, toward Minus infinity (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Minus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

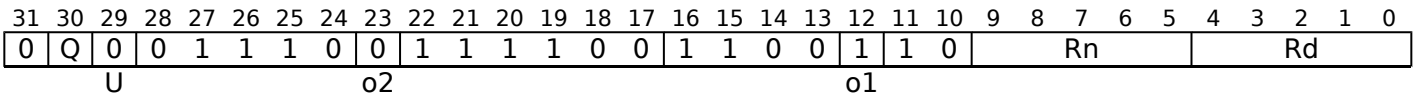
A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

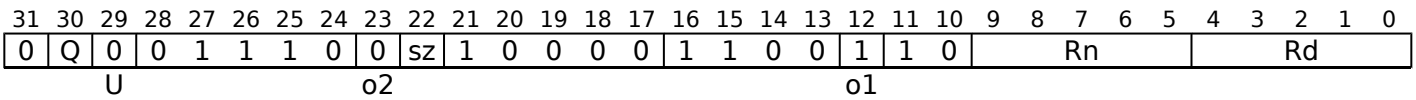
(FEAT_FP16)



FRINTM <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR[]);
```

Single-precision and double-precision



FRINTM <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

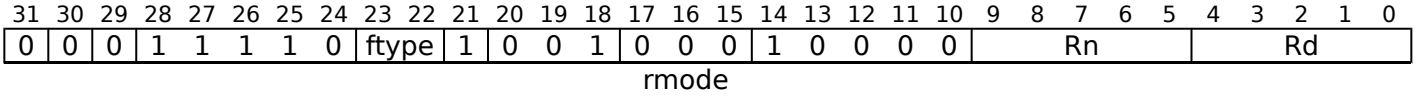
FRINTN (scalar)

Floating-point Round to Integral, to nearest with ties to even (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)

(FEAT_FP16)

```
FRINTN <Hd>, <Hn>
```

Single-precision (ftype == 00)

```
FRINTN <Sd>, <Sn>
```

Double-precision (ftype == 01)

```
FRINTN <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;

FPRounding rounding;
rounding = FPDecodeRounding('00');
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);
bits(esize) operand = V[n, esize];

Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FRINTN (vector)

Floating-point Round to Integral, to nearest with ties to even (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round to Nearest rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

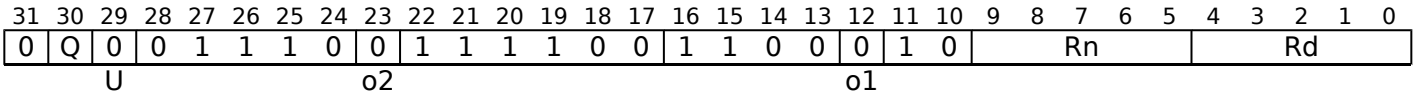
A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

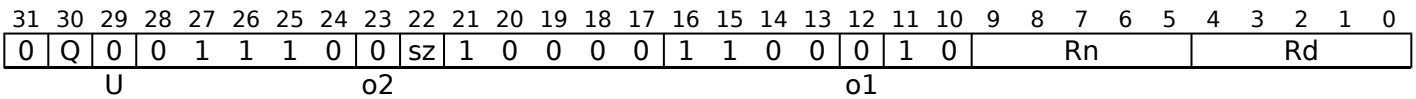
(FEAT_FP16)



FRINTN <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR[]);
```

Single-precision and double-precision



FRINTN <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

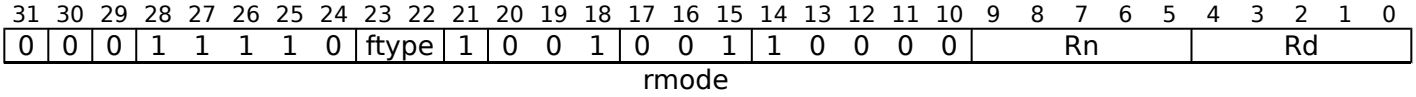
FRINTP (scalar)

Floating-point Round to Integral, toward Plus infinity (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)

(FEAT_FP16)

```
FRINTP <Hd>, <Hn>
```

Single-precision (ftype == 00)

```
FRINTP <Sd>, <Sn>
```

Double-precision (ftype == 01)

```
FRINTP <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;

FPRounding rounding;
rounding = FPDecodeRounding('01');
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);
bits(esize) operand = V[n, esize];

Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FRINTP (vector)

Floating-point Round to Integral, toward Plus infinity (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Plus Infinity rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

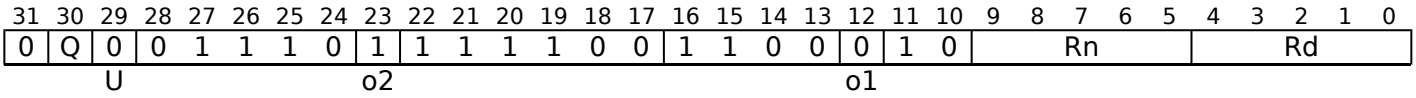
A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

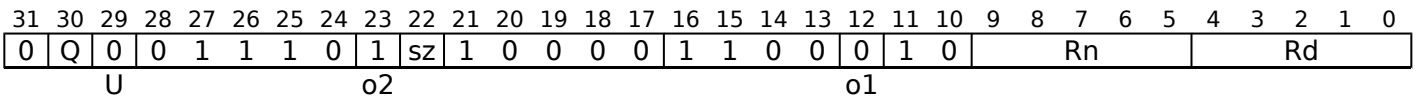
(FEAT_FP16)



FRINTP <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR[]);
```

Single-precision and double-precision



FRINTP <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

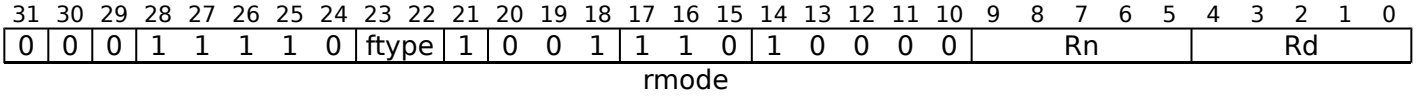
FRINTX (scalar)

Floating-point Round to Integral exact, using current rounding mode (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the rounding mode that is determined by the *FPCR*, and writes the result to the SIMD&FP destination register.

When the result value is not numerically equal to the input value, an Inexact exception is raised. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)

(FEAT_FP16)

```
FRINTX <Hd>, <Hn>
```

Single-precision (ftype == 00)

```
FRINTX <Sd>, <Sn>
```

Double-precision (ftype == 01)

```
FRINTX <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;

FPRounding rounding;
rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);
bits(esize) operand = V[n, esize];

Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, TRUE);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FRINTX (vector)

Floating-point Round to Integral exact, using current rounding mode (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the rounding mode that is determined by the *FPCR*, and writes the result to the SIMD&FP destination register.

When a result value is not numerically equal to the corresponding input value, an Inexact exception is raised. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

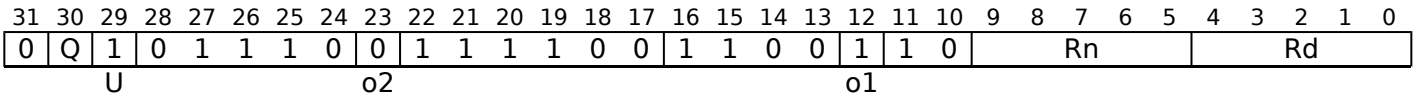
A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

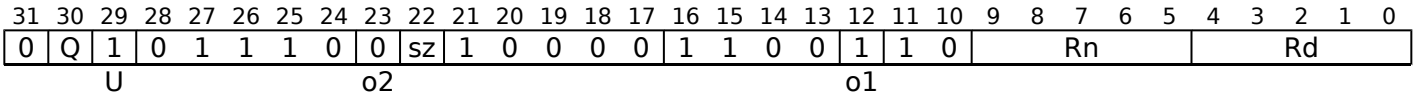
(FEAT_FP16)



FRINTX <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR[]);
```

Single-precision and double-precision



FRINTX <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

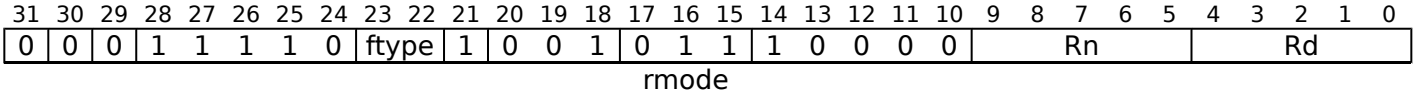
FRINTZ (scalar)

Floating-point Round to Integral, toward Zero (scalar). This instruction rounds a floating-point value in the SIMD&FP source register to an integral floating-point value of the same size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)

(FEAT_FP16)

```
FRINTZ <Hd>, <Hn>
```

Single-precision (ftype == 00)

```
FRINTZ <Sd>, <Sn>
```

Double-precision (ftype == 01)

```
FRINTZ <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;

FPRounding rounding;
rounding = FPDecodeRounding('11');
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);
bits(esize) operand = V[n, esize];

Elem[result, 0, esize] = FPRoundInt(operand, fpcr, rounding, FALSE);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FRINTZ (vector)

Floating-point Round to Integral, toward Zero (vector). This instruction rounds a vector of floating-point values in the SIMD&FP source register to integral floating-point values of the same size using the Round towards Zero rounding mode, and writes the result to the SIMD&FP destination register.

A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

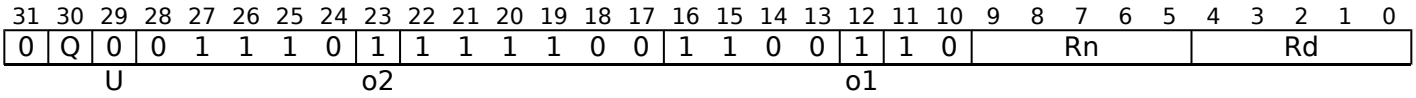
A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

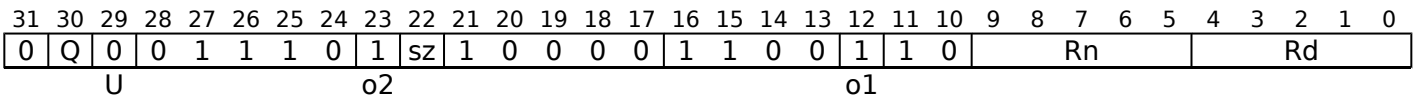
(FEAT_FP16)



FRINTZ <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR[]);
```

Single-precision and double-precision



FRINTZ <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean exact = FALSE;
FPRounding rounding;
case U:o1:o2 of
    when '0xx' rounding = FPDecodeRounding(o1:o2);
    when '100' rounding = FPRounding_TIEAWAY;
    when '101' UNDEFINED;
    when '110' rounding = FPRoundingMode(FPCR[]); exact = TRUE;
    when '111' rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FRSQRTE

Floating-point Reciprocal Square Root Estimate. This instruction calculates an approximate square root for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn					Rd				

FRSQRTE <Hd>, <Hn>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	0	1	1	0	Rn					Rd				

FRSQRTE <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

Vector half precision

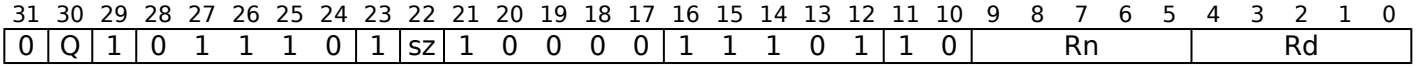
(FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	1	1	1	1	0	0	1	1	1	0	1	1	0	Rn					Rd				

FRSQRTE <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Vector single-precision and double-precision



FRSQRTE <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
if elements == 1 then
    CheckFPEnabled64();
else
    CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRSqrtEstimate(element, fpcr);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FRSQRTS

Floating-point Reciprocal Square Root Step. This instruction multiplies corresponding floating-point values in the vectors of the two source SIMD&FP registers, subtracts each of the products from 3.0, divides these results by 2.0, places the results into a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

(FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	1	0	Rm					0	0	1	1	1	1	Rn					Rd				

FRSQRTS <Hd>, <Hn>, <Hm>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = esize;
integer elements = 1;
```

Scalar single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	1	sz	1	Rm					1	1	1	1	1	1	Rn					Rd				

FRSQRTS <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
```

Vector half precision

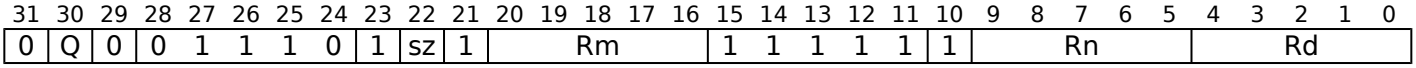
(FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	1	1	0	Rm					0	0	1	1	1	1	Rn					Rd				

FRSQRTS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Vector single-precision and double-precision



FRSQRTS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.

<Hm> Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
if elements == 1 then
    CheckFPEnabled64();
else
    CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPRSqrtStepFused(element1, element2);

V[d, 128] = result;
```

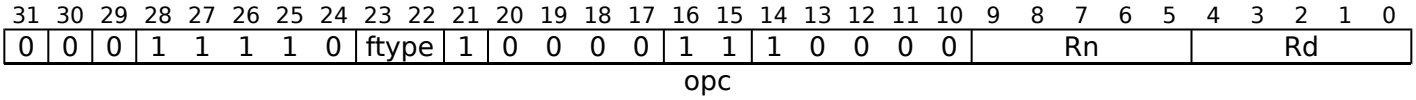
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FSQRT (scalar)

Floating-point Square Root (scalar). This instruction calculates the square root of the value in the SIMD&FP source register and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)

(FEAT_FP16)

```
FSQRT <Hd>, <Hn>
```

Single-precision (ftype == 00)

```
FSQRT <Sd>, <Sn>
```

Double-precision (ftype == 01)

```
FSQRT <Dd>, <Dn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else 0<127:0>;

bits(esize) operand = V[n, esize];

Elem[result, 0, esize] = FPSqrt(operand, fpcr);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FSQRT (vector)

Floating-point Square Root (vector). This instruction calculates the square root for each vector element in the source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

(FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	1	1	1	1	0	0	1	1	1	1	1	1	0	Rn					Rd				

FSQRT <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Single-precision and double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	1	sz	1	0	0	0	0	1	1	1	1	1	1	0	Rn					Rd				

FSQRT <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPSqrt(element, FPCR[]);

V[d, datasize] = result;
```

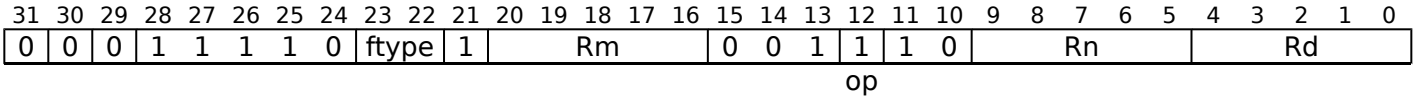
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FSUB (scalar)

Floating-point Subtract (scalar). This instruction subtracts the floating-point value of the second source SIMD&FP register from the floating-point value of the first source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Half-precision (ftype == 11)

(FEAT_FP16)

```
FSUB <Hd>, <Hn>, <Hm>
```

Single-precision (ftype == 00)

```
FSUB <Sd>, <Sn>, <Sm>
```

Double-precision (ftype == 01)

```
FSUB <Dd>, <Dn>, <Dm>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer esize;
case ftype of
    when '00' esize = 32;
    when '01' esize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            esize = 16;
        else
            UNDEFINED;
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hn>	Is the 16-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Hm>	Is the 16-bit name of the second SIMD&FP source register, encoded in the "Rm" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Rn" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPEnabled64();
bits(esize) operand1 = V[n, esize];
bits(esize) operand2 = V[m, esize];

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
bits(128) result = if merge then V[n, 128] else Zeros(128);

Elem[result, 0, esize] = FPSub(operand1, operand2, fpcr);
V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FSUB (vector)

Floating-point Subtract (vector). This instruction subtracts the elements in the vector in the second source SIMD&FP register, from the corresponding elements in the vector in the first source SIMD&FP register, places each result into elements of a vector, and writes the vector to the destination SIMD&FP register.

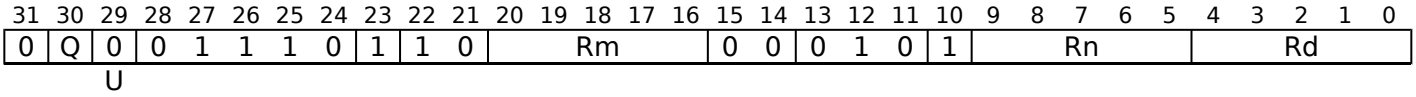
This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Half-precision and Single-precision and double-precision

Half-precision

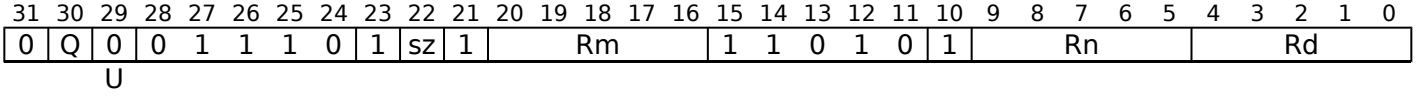
(FEAT_FP16)



FSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');
```

Single-precision and double-precision



FSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean abs = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];

bits(esize) element1;
bits(esize) element2;
bits(esize) diff;
FPCRType fpcr = FPCR[];
bits(datasize) result;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    diff = FPSub(element1, element2, fpcr);
    Elem[result, e, esize] = if abs then FPAbs(diff) else diff;

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

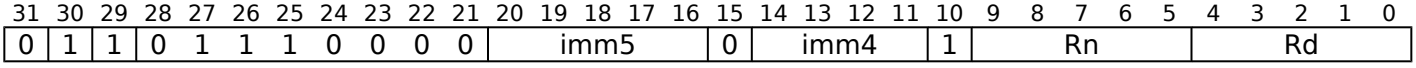
INS (element)

Insert vector element from another vector element. This instruction copies the vector element of the source SIMD&FP register to the specified vector element of the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (element).



INS <Vd>.<Ts>[<index1>], <Vn>.<Ts>[<index2>]

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;

integer dst_index = UInt(imm5<4:size+1>);
integer src_index = UInt(imm4<3:size>);
integer idxdsize = if imm4<3> == '1' then 128 else 64;
// imm4<size-1:0> is IGNORED

integer esize = 8 << size;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ts> Is an element size specifier, encoded in “imm5”:

imm5	<Ts>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<index1> Is the destination element index encoded in “imm5”:

imm5	<index1>
x0000	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>
xx100	imm5<4:3>
x1000	imm5<4>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<index2> Is the source element index encoded in “imm5:imm4”:

imm5	<index2>
x0000	RESERVED
xxxx1	imm4<3:0>
xxx10	imm4<3:1>
xx100	imm4<3:2>
x1000	imm4<3>

Unspecified bits in "imm4" are ignored but should be set to zero by an assembler.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(idxdsize) operand = V[n, idxdsize];
bits(128) result;

result = V[d, 128];
Elem[result, dst_index, esize] = Elem[operand, src_index, esize];
V[d, 128] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

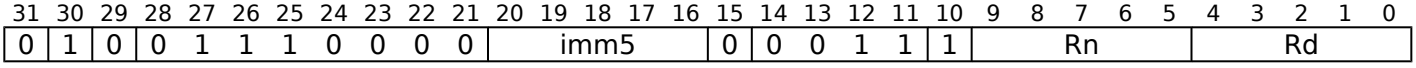
INS (general)

Insert vector element from general-purpose register. This instruction copies the contents of the source general-purpose register to the specified vector element in the destination SIMD&FP register.

This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (from general).



INS <Vd>.<Ts>[<index>], <R><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);

if size > 3 then UNDEFINED;
integer index = UInt(imm5<4:size+1>);

integer esize = 8 << size;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ts> Is an element size specifier, encoded in “imm5”:

imm5	<Ts>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<index> Is the element index encoded in “imm5”:

imm5	<index>
x0000	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>
xx100	imm5<4:3>
x1000	imm5<4>

<R> Is the width specifier for the general-purpose source register, encoded in “imm5”:

imm5	<R>
x0000	RESERVED
xxxx1	W
xxx10	W
xx100	W
x1000	X

<n> Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(esize) element = X[n, esize];
bits(128) result;

result = V[d, 128];
Elem[result, index, esize] = element;
V[d, 128] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

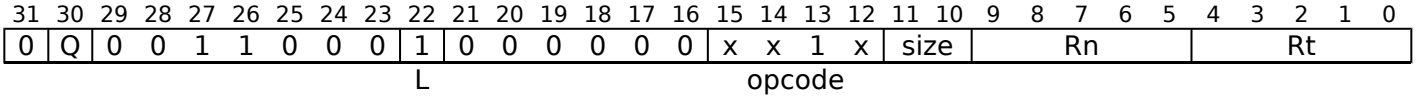
LD1 (multiple structures)

Load multiple single-element structures to one, two, three, or four registers. This instruction loads multiple single-element structures from memory and writes the result to one, two, three, or four SIMD&FP registers.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset



One register (opcode == 0111)

```
LD1 { <Vt>.<T> }, [<Xn|SP>]
```

Two registers (opcode == 1010)

```
LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

Three registers (opcode == 0110)

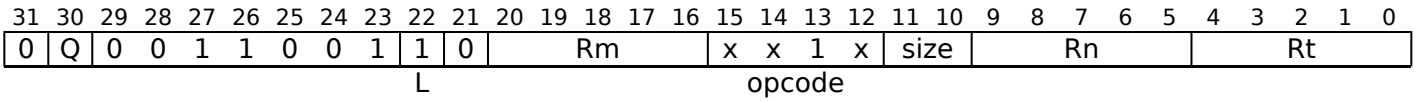
```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

Four registers (opcode == 0010)

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index



One register, immediate offset (Rm == 11111 && opcode == 0111)

```
LD1 { <Vt>.<T> }, [<Xn|SP>], <imm>
```

One register, register offset (Rm != 11111 && opcode == 0111)

```
LD1 { <Vt>.<T> }, [<Xn|SP>], <Xm>
```

Two registers, immediate offset (Rm == 11111 && opcode == 1010)

```
LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Two registers, register offset (Rm != 11111 && opcode == 1010)

```
LD1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

Three registers, immediate offset (Rm == 11111 && opcode == 0110)

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Three registers, register offset (Rm != 11111 && opcode == 0110)

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

Four registers, immediate offset (Rm == 11111 && opcode == 0010)

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Four registers, register offset (Rm != 11111 && opcode == 0010)

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> For the one register, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#8
1	#16

For the two registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#16
1	#32

For the three registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#24
1	#48

For the four registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#32
1	#64

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;    // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
    otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt, datasize];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt, datasize] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

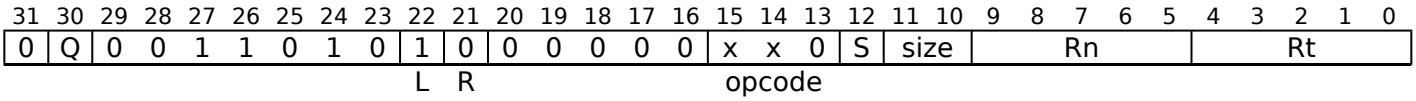
LD1 (single structure)

Load one single-element structure to one lane of one register. This instruction loads a single-element structure from memory and writes the result to the specified lane of the SIMD&FP register without affecting the other bits of the register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset



8-bit (opcode == 000)

```
LD1 { <Vt>.B }[<index>], [<Xn|SP>]
```

16-bit (opcode == 010 && size == x0)

```
LD1 { <Vt>.H }[<index>], [<Xn|SP>]
```

32-bit (opcode == 100 && size == 00)

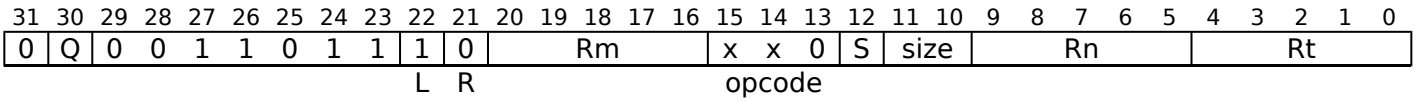
```
LD1 { <Vt>.S }[<index>], [<Xn|SP>]
```

64-bit (opcode == 100 && S == 0 && size == 01)

```
LD1 { <Vt>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index



8-bit, immediate offset (Rm == 11111 && opcode == 000)

```
LD1 { <Vt>.B }[<index>], [<Xn|SP>], #1
```

8-bit, register offset (Rm != 11111 && opcode == 000)

```
LD1 { <Vt>.B }[<index>], [<Xn|SP>], <Xm>
```

16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)

```
LD1 { <Vt>.H }[<index>], [<Xn|SP>], #2
```

16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)

```
LD1 { <Vt>.H }[<index>], [<Xn|SP>], <Xm>
```

32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)

```
LD1 { <Vt>.S }[<index>], [<Xn|SP>], #4
```

32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)

```
LD1 { <Vt>.S }[<index>], [<Xn|SP>], <Xm>
```

64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)

```
LD1 { <Vt>.D }[<index>], [<Xn|SP>], #8
```

64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)

```
LD1 { <Vt>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t, 128] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

Operational information

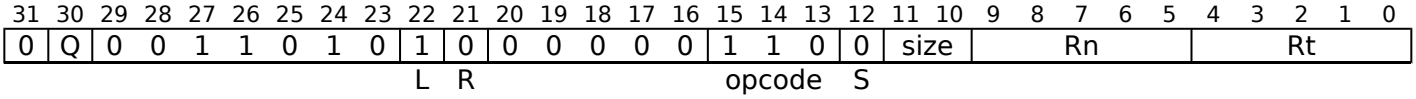
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1R

Load one single-element structure and Replicate to all lanes (of one register). This instruction loads a single-element structure from memory and replicates the structure to all the lanes of the SIMD&FP register.
Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.
It has encodings from 2 classes: No offset and Post-index

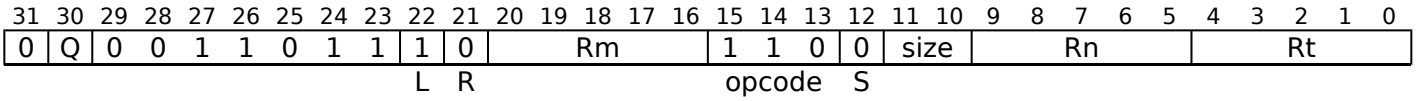
No offset



LD1R { <Vt>.<T> }, [<Xn|SP>]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index



Immediate offset (Rm == 11111)

```
LD1R { <Vt>.<T> }, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```
LD1R { <Vt>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in “size”:

size	<imm>
00	#1
01	#2
10	#4
11	#8

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t, 128] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD2 (multiple structures)

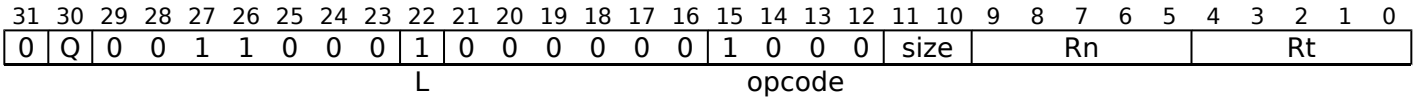
Load multiple 2-element structures to two registers. This instruction loads multiple 2-element structures from memory and writes the result to the two SIMD&FP registers, with de-interleaving.

For an example of de-interleaving, see `LD3 (multiple structures)`.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

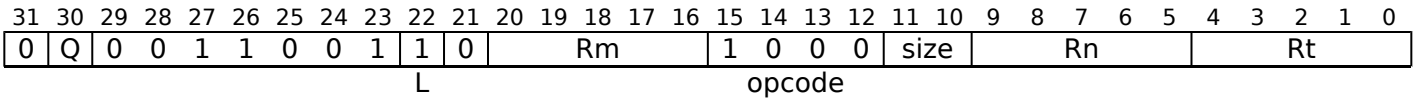
No offset



LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index



Immediate offset (Rm == 11111)

```
LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```
LD2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#16
1	#32

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;    // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
    otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt, datasize];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt, datasize] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

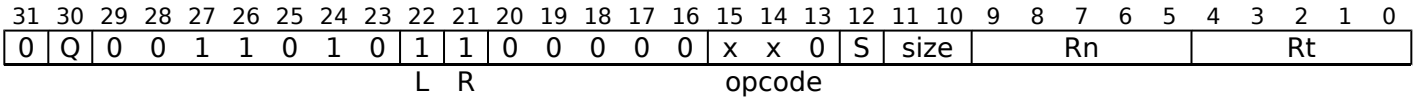
LD2 (single structure)

Load single 2-element structure to one lane of two registers. This instruction loads a 2-element structure from memory and writes the result to the corresponding elements of the two SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset



8-bit (opcode == 000)

```
LD2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>]
```

16-bit (opcode == 010 && size == x0)

```
LD2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>]
```

32-bit (opcode == 100 && size == 00)

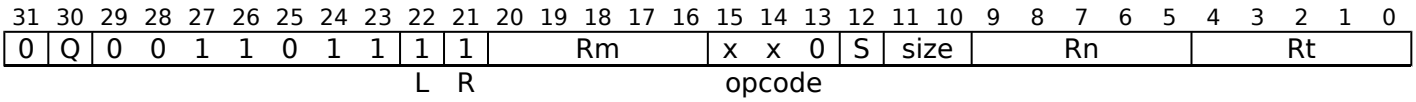
```
LD2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>]
```

64-bit (opcode == 100 && S == 0 && size == 01)

```
LD2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index



8-bit, immediate offset (Rm == 11111 && opcode == 000)

```
LD2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], #2
```

8-bit, register offset (Rm != 11111 && opcode == 000)

```
LD2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], <Xm>
```

16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)

```
LD2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], #4
```

16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)

```
LD2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], <Xm>
```

32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)

```
LD2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], #8
```

32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)

```
LD2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], <Xm>
```

64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)

```
LD2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], #16
```

64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)

```
LD2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t, 128] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

Operational information

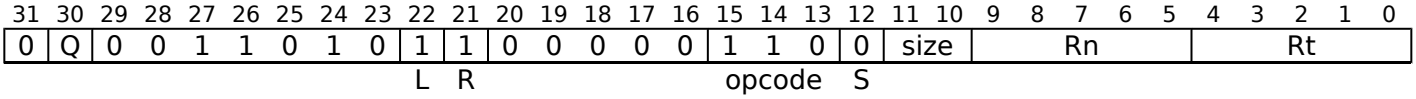
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD2R

Load single 2-element structure and Replicate to all lanes of two registers. This instruction loads a 2-element structure from memory and replicates the structure to all the lanes of the two SIMD&FP registers.
Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.
It has encodings from 2 classes: No offset and Post-index

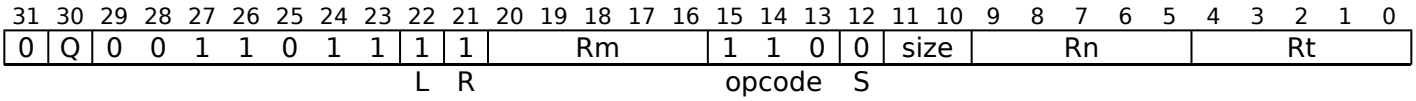
No offset



LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index



Immediate offset (Rm == 11111)

```
LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```
LD2R { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in “size”:

size	<imm>
00	#2
01	#4
10	#8
11	#16

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t, 128] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

Operational information

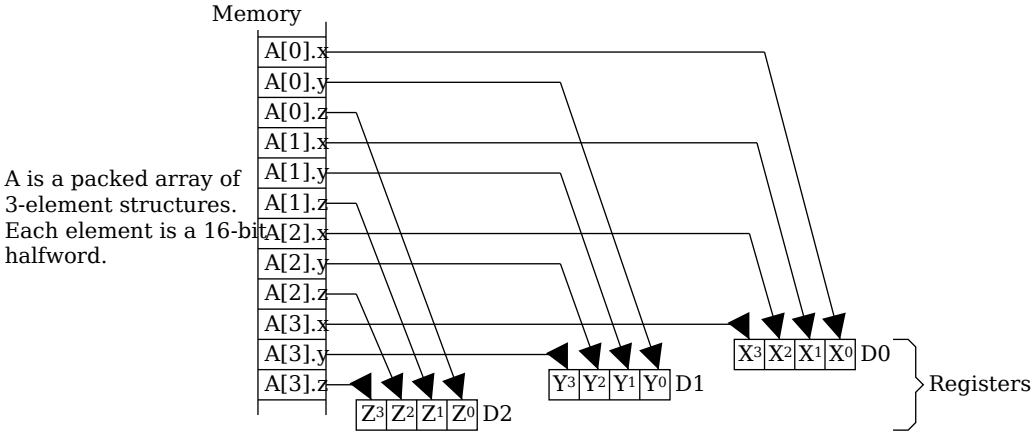
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD3 (multiple structures)

Load multiple 3-element structures to three registers. This instruction loads multiple 3-element structures from memory and writes the result to the three SIMD&FP registers, with de-interleaving.

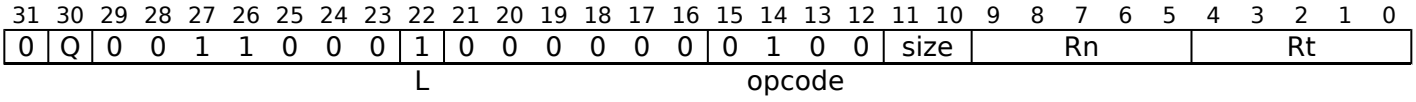
The following figure shows an example of the operation of de-interleaving of a LD3.16 (multiple 3-element structures) instruction:.



Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

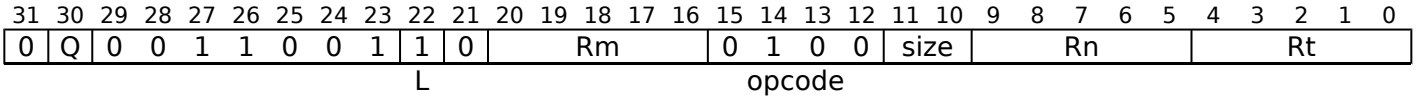
No offset



LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index



Immediate offset (Rm == 11111)

```
LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```
LD3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#24
1	#48

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;    // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
    otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt, datasize];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt, datasize] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

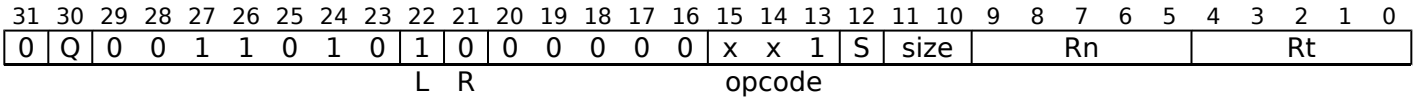
LD3 (single structure)

Load single 3-element structure to one lane of three registers. This instruction loads a 3-element structure from memory and writes the result to the corresponding elements of the three SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset



8-bit (opcode == 001)

```
LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>]
```

16-bit (opcode == 011 && size == x0)

```
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>]
```

32-bit (opcode == 101 && size == 00)

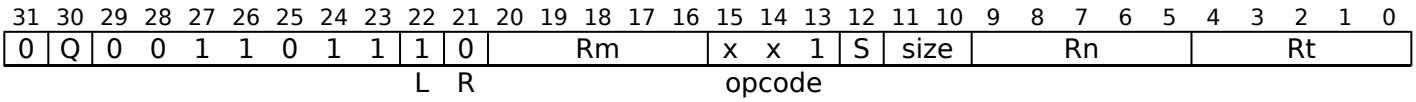
```
LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>]
```

64-bit (opcode == 101 && S == 0 && size == 01)

```
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index



8-bit, immediate offset (Rm == 11111 && opcode == 001)

```
LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], #3
```

8-bit, register offset (Rm != 11111 && opcode == 001)

```
LD3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], <Xm>
```

16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

```
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], #6
```

16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

```
LD3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], <Xm>
```

32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

```
LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], #12
```

32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

```
LD3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], <Xm>
```

64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

```
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], #24
```

64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

```
LD3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3>	Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t, 128] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

Operational information

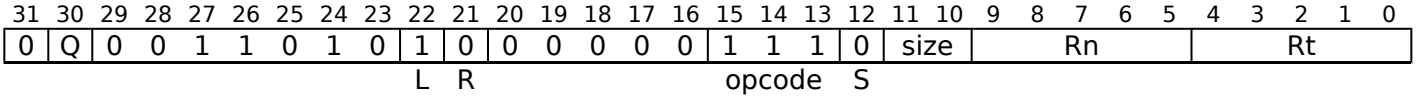
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD3R

Load single 3-element structure and Replicate to all lanes of three registers. This instruction loads a 3-element structure from memory and replicates the structure to all the lanes of the three SIMD&FP registers.
Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.
It has encodings from 2 classes: No offset and Post-index

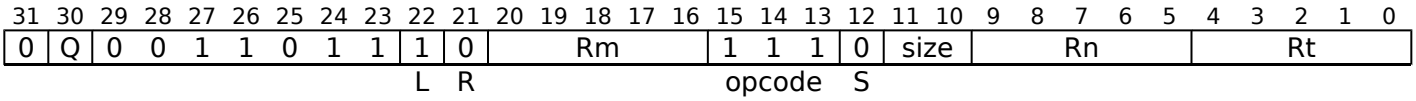
No offset



LD3R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index



Immediate offset (Rm == 11111)

LD3R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>

Register offset (Rm != 11111)

LD3R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in “size”:

size	<imm>
00	#3
01	#6
10	#12
11	#24

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t, 128] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD4 (multiple structures)

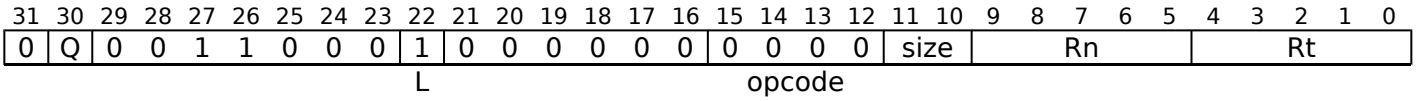
Load multiple 4-element structures to four registers. This instruction loads multiple 4-element structures from memory and writes the result to the four SIMD&FP registers, with de-interleaving.

For an example of de-interleaving, see `LD3 (multiple structures)`.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

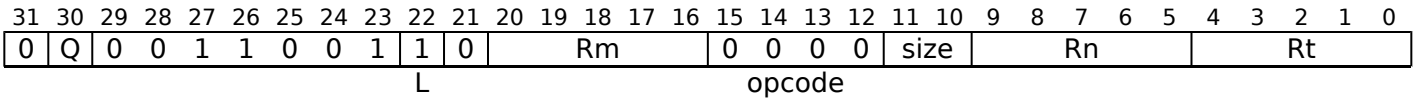
No offset



LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index



Immediate offset (Rm == 11111)

```
LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```
LD4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#32
1	#64

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;    // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
    otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt, datasize];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt, datasize] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

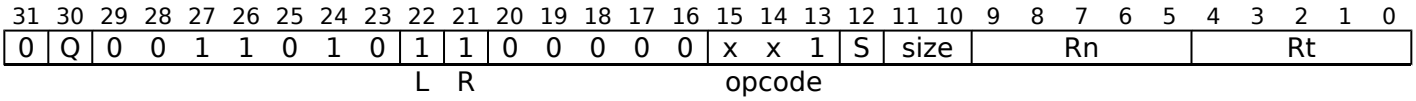
LD4 (single structure)

Load single 4-element structure to one lane of four registers. This instruction loads a 4-element structure from memory and writes the result to the corresponding elements of the four SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset



8-bit (opcode == 001)

```
LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>]
```

16-bit (opcode == 011 && size == x0)

```
LD4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>]
```

32-bit (opcode == 101 && size == 00)

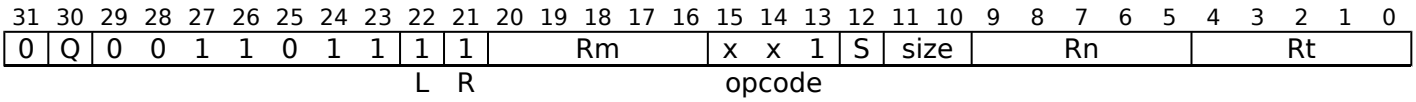
```
LD4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>]
```

64-bit (opcode == 101 && S == 0 && size == 01)

```
LD4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index



8-bit, immediate offset (Rm == 11111 && opcode == 001)

```
LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], #4
```

8-bit, register offset (Rm != 11111 && opcode == 001)

```
LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], <Xm>
```

16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

```
LD4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], #8
```

16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

```
LD4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], <Xm>
```

32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

```
LD4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], #16
```

32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

```
LD4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], <Xm>
```

64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

```
LD4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], #32
```

64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

```
LD4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3>	Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<Vt4>	Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t, 128] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

Operational information

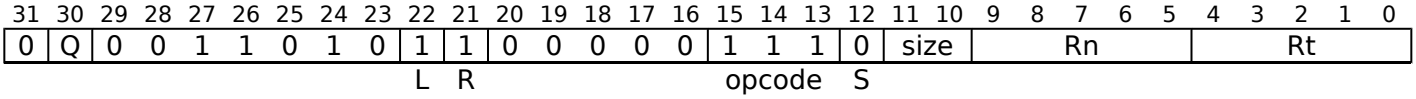
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD4R

Load single 4-element structure and Replicate to all lanes of four registers. This instruction loads a 4-element structure from memory and replicates the structure to all the lanes of the four SIMD&FP registers.
Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.
It has encodings from 2 classes: No offset and Post-index

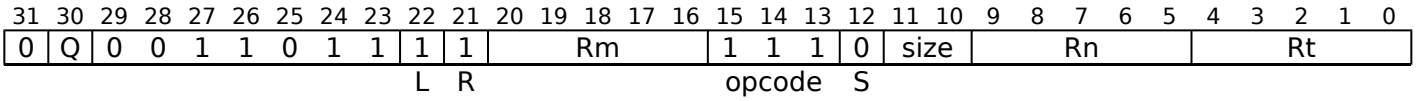
No offset



LD4R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index



Immediate offset (Rm == 11111)

```
LD4R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```
LD4R { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in “size”:

size	<imm>
00	#4
01	#8
10	#16
11	#32

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t, 128] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

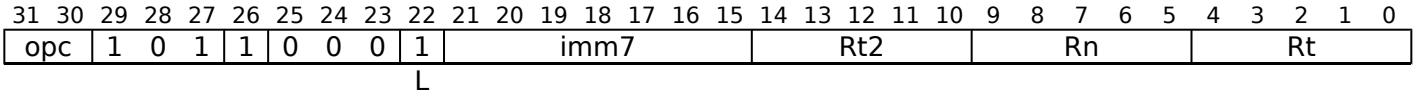
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDNP (SIMD&FP)

Load Pair of SIMD&FP registers, with Non-temporal hint. This instruction loads a pair of SIMD&FP registers from memory, issuing a hint to the memory system that the access is non-temporal. The address that is used for the load is calculated from a base register value and an optional immediate offset.

For information about non-temporal pair instructions, see *Load/Store SIMD and Floating-point Non-temporal pair*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



32-bit (opc == 00)

```
LDNP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

64-bit (opc == 01)

```
LDNP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

128-bit (opc == 10)

```
LDNP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

```
// Empty.
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDNP (SIMD&FP)*.

Assembler Symbols

<Dt1>	Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt2>	Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Qt1>	Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt2>	Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<St1>	Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<St2>	Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4. For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8. For the 128-bit variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Operation

```
CheckFPEnabled64();
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data1 = Mem[address, dbytes, AccType_VECSTREAM];
data2 = Mem[address+dbytes, dbytes, AccType_VECSTREAM];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
V[t, datasize] = data1;
V[t2, datasize] = data2;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

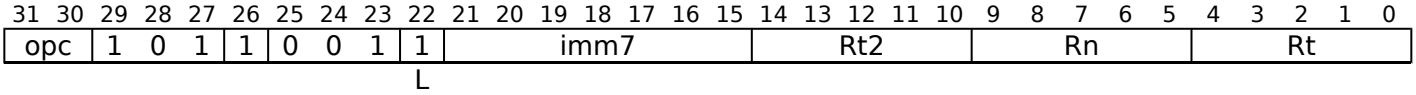
LDP (SIMD&FP)

Load Pair of SIMD&FP registers. This instruction loads a pair of SIMD&FP registers from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: Post-index , Pre-index and Signed offset

Post-index



32-bit (opc == 00)

```
LDP <St1>, <St2>, [<Xn|SP>], #<imm>
```

64-bit (opc == 01)

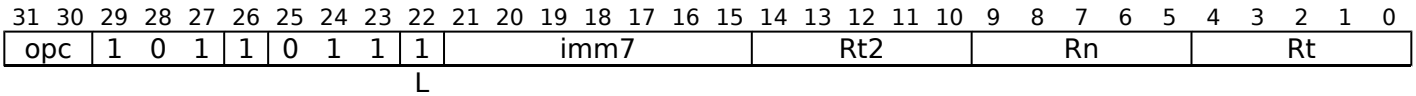
```
LDP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>
```

128-bit (opc == 10)

```
LDP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

Pre-index



32-bit (opc == 00)

```
LDP <St1>, <St2>, [<Xn|SP>, #<imm>]!
```

64-bit (opc == 01)

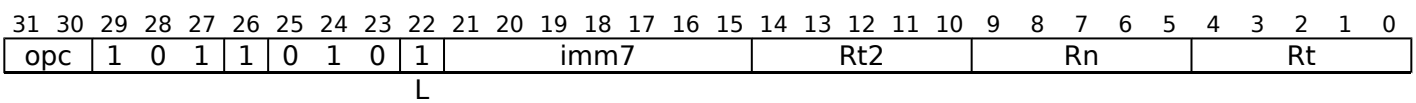
```
LDP <Dt1>, <Dt2>, [<Xn|SP>, #<imm>]!
```

128-bit (opc == 10)

```
LDP <Qt1>, <Qt2>, [<Xn|SP>, #<imm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

Signed offset



32-bit (opc == 00)

```
LDP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

64-bit (opc == 01)

```
LDP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

128-bit (opc == 10)

```
LDP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDP (SIMD&FP)*.

Assembler Symbols

<Dt1>	Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt2>	Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Qt1>	Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt2>	Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<St1>	Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<St2>	Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4. For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4. For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8. For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8. For the 128-bit post-index and 128-bit pre-index variant: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field as <imm>/16. For the 128-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;

boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF   UNDEFINED;
        when Constraint_NOP     EndOfInstruction();
```

Operation

```
CheckFPEnabled64();
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

data1 = Mem[address, dbytes, AccType_VEC];
data2 = Mem[address+dbytes, dbytes, AccType_VEC];
if rt_unknown then
    data1 = bits(datasize) UNKNOWN;
    data2 = bits(datasize) UNKNOWN;
V[t, datasize] = data1;
V[t2, datasize] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

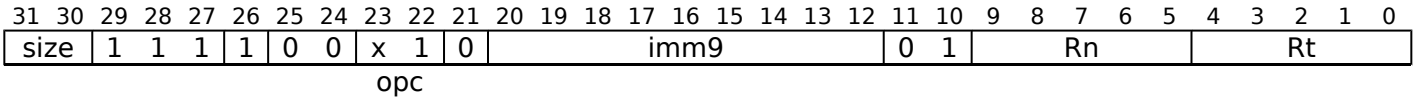
LDR (immediate, SIMD&FP)

Load SIMD&FP Register (immediate offset). This instruction loads an element from memory, and writes the result as a scalar to the SIMD&FP register. The address that is used for the load is calculated from a base register value, a signed immediate offset, and an optional offset that is a multiple of the element size.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index



8-bit (size == 00 && opc == 01)

```
LDR <Bt>, [<Xn|SP>], #<simm>
```

16-bit (size == 01 && opc == 01)

```
LDR <Ht>, [<Xn|SP>], #<simm>
```

32-bit (size == 10 && opc == 01)

```
LDR <St>, [<Xn|SP>], #<simm>
```

64-bit (size == 11 && opc == 01)

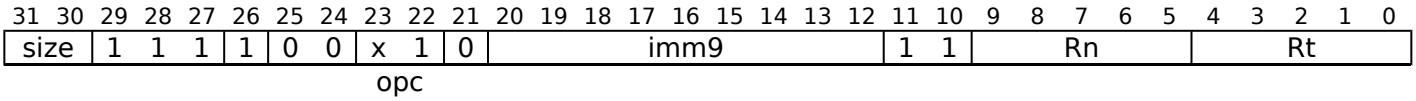
```
LDR <Dt>, [<Xn|SP>], #<simm>
```

128-bit (size == 00 && opc == 11)

```
LDR <Qt>, [<Xn|SP>], #<simm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index



8-bit (size == 00 && opc == 01)

```
LDR <Bt>, [<Xn|SP>, #<simm>]!
```

16-bit (size == 01 && opc == 01)

```
LDR <Ht>, [<Xn|SP>, #<simm>]!
```

32-bit (size == 10 && opc == 01)

```
LDR <St>, [<Xn|SP>, #<simm>]!
```

64-bit (size == 11 && opc == 01)

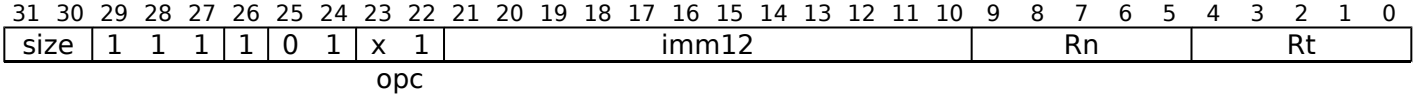
```
LDR <Dt>, [<Xn|SP>, #<simm>]!
```

128-bit (size == 00 && opc == 11)

```
LDR <Qt>, [<Xn|SP>, #<simm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



8-bit (size == 00 && opc == 01)

```
LDR <Bt>, [<Xn|SP>{, #<pimm>}]
```

16-bit (size == 01 && opc == 01)

```
LDR <Ht>, [<Xn|SP>{, #<pimm>}]
```

32-bit (size == 10 && opc == 01)

```
LDR <St>, [<Xn|SP>{, #<pimm>}]
```

64-bit (size == 11 && opc == 01)

```
LDR <Dt>, [<Xn|SP>{, #<pimm>}]
```

128-bit (size == 00 && opc == 11)

```
LDR <Qt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

Assembler Symbols

<Bt>	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht>	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	For the 8-bit variant: is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field. For the 16-bit variant: is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2. For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4. For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8. For the 128-bit variant: is the optional positive immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0 and encoded in the "imm12" field as <pimm>/16.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);
```

Operation

```
CheckFPEnabled64();
bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t, datasize];
        Mem[address, datasize DIV 8, AccType_VEC] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, AccType_VEC];
        V[t, datasize] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

Operational information

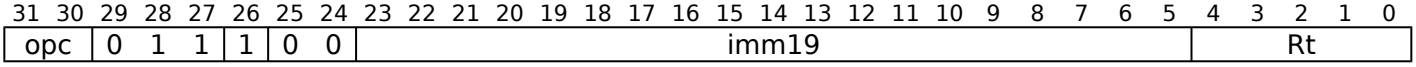
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDR (literal, SIMD&FP)

Load SIMD&FP Register (PC-relative literal). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from the PC value and an immediate offset.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



32-bit (opc == 00)

```
LDR <St>, <label>
```

64-bit (opc == 01)

```
LDR <Dt>, <label>
```

128-bit (opc == 10)

```
LDR <Qt>, <label>
```

```
integer t = UInt(Rt);
integer size;
bits(64) offset;

case opc of
    when '00'
        size = 4;
    when '01'
        size = 8;
    when '10'
        size = 16;
    when '11'
        UNDEFINED;

offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

<Dt>	Is the 64-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.
<label>	Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```
bits(64) address = PC[] + offset;
bits(size*8) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(FALSE);

CheckFPEnabled64();

data = Mem[address, size, AccType_VEC];
V[t, size*8] = data;
```

Operational information

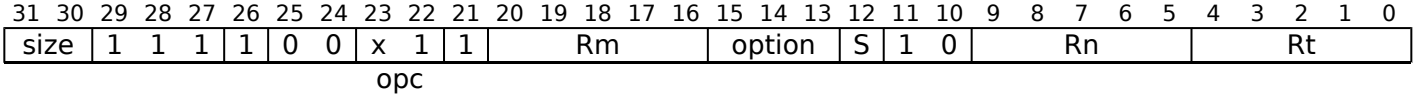
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDR (register, SIMD&FP)

Load SIMD&FP Register (register offset). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from a base register value and an offset register value. The offset can be optionally shifted and extended.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



8-bit (size == 00 && opc == 01 && option != 011)

```
LDR <Bt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

8-bit (size == 00 && opc == 01 && option == 011)

```
LDR <Bt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

16-bit (size == 01 && opc == 01)

```
LDR <Ht>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

32-bit (size == 10 && opc == 01)

```
LDR <St>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

64-bit (size == 11 && opc == 01)

```
LDR <Dt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

128-bit (size == 00 && opc == 11)

```
LDR <Qt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

<Bt>	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht>	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Wm>	When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
<Xm>	When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> For the 8-bit variant: is the index extend specifier, encoded in “option”:

option	<extend>
010	UXTW
110	SXTW
111	SXTX

For the 128-bit, 16-bit, 32-bit and 64-bit variant: is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount> For the 8-bit variant: is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

For the 16-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#1

For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#2

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#3

For the 128-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#4

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift, 64);
CheckFPEnabled64();
bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t, datasize];
        Mem[address, datasize DIV 8, AccType_VEC] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, AccType_VEC];
        V[t, datasize] = data;
```

Operational information

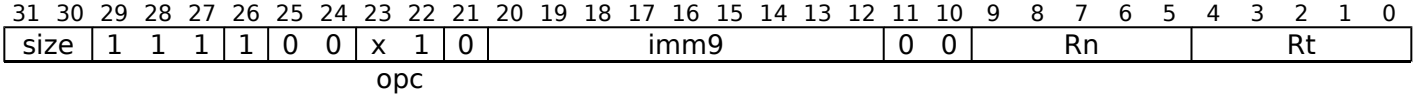
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDUR (SIMD&FP)

Load SIMD&FP Register (unscaled offset). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



8-bit (size == 00 && opc == 01)

```
LDUR <Bt>, [<Xn|SP>{, #<simm>}]
```

16-bit (size == 01 && opc == 01)

```
LDUR <Ht>, [<Xn|SP>{, #<simm>}]
```

32-bit (size == 10 && opc == 01)

```
LDUR <St>, [<Xn|SP>{, #<simm>}]
```

64-bit (size == 11 && opc == 01)

```
LDUR <Dt>, [<Xn|SP>{, #<simm>}]
```

128-bit (size == 00 && opc == 11)

```
LDUR <Qt>, [<Xn|SP>{, #<simm>}]
```

```
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Bt>	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht>	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);
```

Operation

```
CheckFPEnabled64();
bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t, datasize];
        Mem[address, datasize DIV 8, AccType_VEC] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, AccType_VEC];
        V[t, datasize] = data;
```

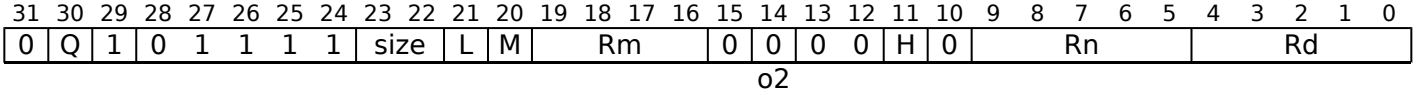
Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MLA (by element)

Multiply-Add to accumulator (vector, by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



MLA <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1*element2)<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

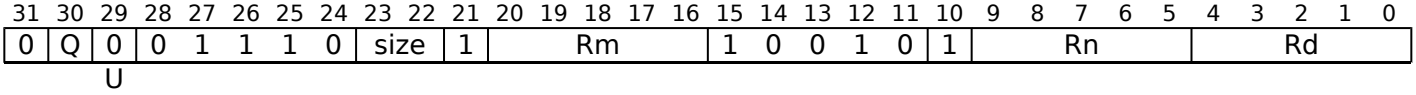
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MLA (vector)

Multiply-Add to accumulator (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, and accumulates the results with the vector elements of the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



MLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    product = (UInt(element1)*UInt(element2))<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

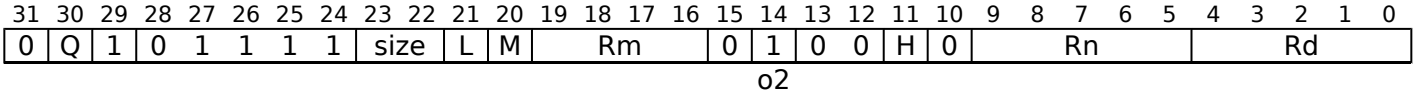
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MLS (by element)

Multiply-Subtract from accumulator (vector, by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and subtracts the results from the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



MLS <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1*element2)<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;
V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

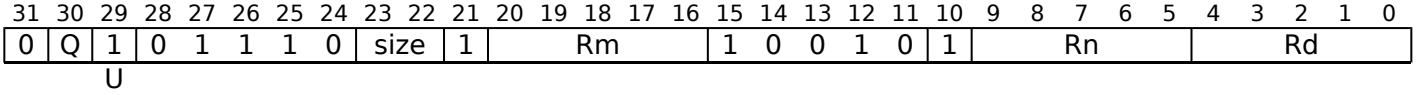
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MLS (vector)

Multiply-Subtract from accumulator (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



MLS <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean sub_op = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    product = (UInt(element1)*UInt(element2))<esize-1:0>;
    if sub_op then
        Elem[result, e, esize] = Elem[operand3, e, esize] - product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize] + product;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

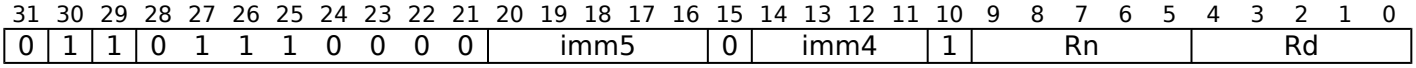
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MOV (element)

Move vector element to another vector element. This instruction copies the vector element of the source SIMD&FP register to the specified vector element of the destination SIMD&FP register.
This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.
Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of INS (element). This means:

- The encodings in this description are named to match the encodings of INS (element).
- The description of INS (element) gives the operational pseudocode for this instruction.



MOV <Vd>.<Ts>[<index1>], <Vn>.<Ts>[<index2>]

is equivalent to

INS <Vd>.<Ts>[<index1>], <Vn>.<Ts>[<index2>]

and is always the preferred disassembly.

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ts> Is an element size specifier, encoded in “imm5”:

imm5	<Ts>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<index1> Is the destination element index encoded in “imm5”:

imm5	<index1>
x0000	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>
xx100	imm5<4:3>
x1000	imm5<4>

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<index2> Is the source element index encoded in “imm5:imm4”:

imm5	<index2>
x0000	RESERVED
xxxx1	imm4<3:0>
xxx10	imm4<3:1>
xx100	imm4<3:2>
x1000	imm4<3>

Unspecified bits in "imm4" are ignored but should be set to zero by an assembler.

Operation

The description of INS (element) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MOV (from general)

Move general-purpose register to a vector element. This instruction copies the contents of the source general-purpose register to the specified vector element in the destination SIMD&FP register.
This instruction can insert data into individual elements within a SIMD&FP register without clearing the remaining bits to zero.
Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of INS (general). This means:

- The encodings in this description are named to match the encodings of INS (general).
- The description of INS (general) gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	0	0	imm5					0	0	0	1	1	1	Rn					Rd				

MOV <Vd>.<Ts>[<index>], <R><n>

is equivalent to

INS <Vd>.<Ts>[<index>], <R><n>

and is always the preferred disassembly.

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ts> Is an element size specifier, encoded in “imm5”:

imm5	<Ts>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<index> Is the element index encoded in “imm5”:

imm5	<index>
x0000	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>
xx100	imm5<4:3>
x1000	imm5<4>

<R> Is the width specifier for the general-purpose source register, encoded in “imm5”:

imm5	<R>
x0000	RESERVED
xxxx1	W
xxx10	W
xx100	W
x1000	X

<n> Is the number [0-30] of the general-purpose source register or ZR (31), encoded in the "Rn" field.

Operation

The description of INS (general) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

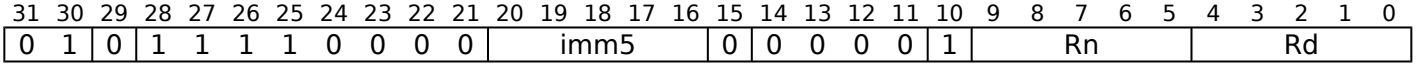
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MOV (scalar)

Move vector element to scalar. This instruction duplicates the specified vector element in the SIMD&FP source register into a scalar, and writes the result to the SIMD&FP destination register.
Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of DUP (element). This means:

- The encodings in this description are named to match the encodings of DUP (element).
- The description of DUP (element) gives the operational pseudocode for this instruction.



MOV <V><d>, <Vn>.<T>[<index>]

is equivalent to

DUP <V><d>, <Vn>.<T>[<index>]

and is always the preferred disassembly.

Assembler Symbols

<V> Is the destination width specifier, encoded in “imm5”:

imm5	<V>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is the element width specifier, encoded in “imm5”:

imm5	<T>
x0000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D

<index> Is the element index encoded in “imm5”:

imm5	<index>
x0000	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>
xx100	imm5<4:3>
x1000	imm5<4>

Operation

The description of DUP (element) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

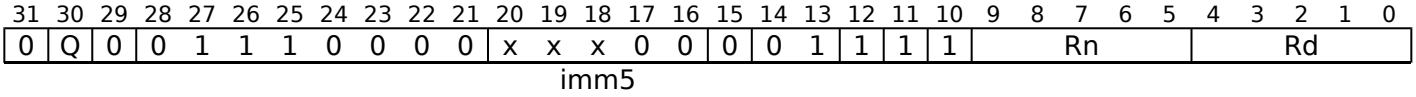
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MOV (to general)

Move vector element to general-purpose register. This instruction reads the unsigned integer from the source SIMD&FP register, zero-extends it to form a 32-bit or 64-bit value, and writes the result to the destination general-purpose register.
Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of UMOV. This means:

- The encodings in this description are named to match the encodings of UMOV.
- The description of UMOV gives the operational pseudocode for this instruction.



32-bit (Q == 0 && imm5 == xx100)

```
MOV <Wd>, <Vn>.S[<index>]
```

is equivalent to

```
UMOV <Wd>, <Vn>.S[<index>]
```

and is always the preferred disassembly.

64-bit (Q == 1 && imm5 == x1000)

```
MOV <Xd>, <Vn>.D[<index>]
```

is equivalent to

```
UMOV <Xd>, <Vn>.D[<index>]
```

and is always the preferred disassembly.

Assembler Symbols

<Wd>	Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Xd>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.
<Vn>	Is the name of the SIMD&FP source register, encoded in the "Rn" field.
<index>	For the 32-bit variant: is the element index encoded in "imm5<4:3>". For the 64-bit variant: is the element index encoded in "imm5<4>".

Operation

The description of UMOV gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

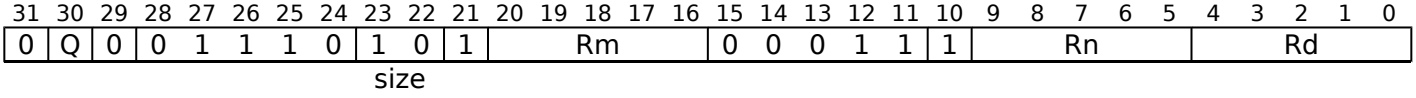
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MOV (vector)

Move vector. This instruction copies the vector in the source SIMD&FP register into the destination SIMD&FP register.
Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of ORR (vector, register). This means:

- The encodings in this description are named to match the encodings of ORR (vector, register).
- The description of ORR (vector, register) gives the operational pseudocode for this instruction.



MOV <Vd>.<T>, <Vn>.<T>

is equivalent to

ORR <Vd>.<T>, <Vn>.<T>, <Vn>.<T>

and is the preferred disassembly when `Rm == Rn`.

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

Operation

The description of ORR (vector, register) gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

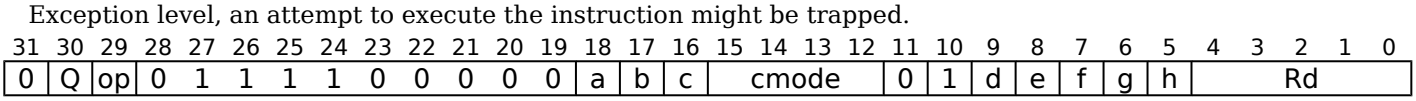
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MOVI

Move Immediate (vector). This instruction places an immediate constant into every vector element of the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



8-bit (op == 0 && cmode == 1110)

```
MOVI <Vd>.<T>, #<imm8>{, LSL #0}
```

16-bit shifted immediate (op == 0 && cmode == 10x0)

```
MOVI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

32-bit shifted immediate (op == 0 && cmode == 0xx0)

```
MOVI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

32-bit shifting ones (op == 0 && cmode == 110x)

```
MOVI <Vd>.<T>, #<imm8>, MSL #<amount>
```

64-bit scalar (Q == 0 && op == 1 && cmode == 1110)

```
MOVI <Dd>, #<imm>
```

64-bit vector (Q == 1 && op == 1 && cmode == 1110)

```
MOVI <Vd>.2D, #<imm>
```

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
    when '0xx00' operation = ImmediateOp_MOVI;
    when '0xx01' operation = ImmediateOp_MVNI;
    when '0xx10' operation = ImmediateOp_ORR;
    when '0xx11' operation = ImmediateOp_BIC;
    when '10x00' operation = ImmediateOp_MOVI;
    when '10x01' operation = ImmediateOp_MVNI;
    when '10x10' operation = ImmediateOp_ORR;
    when '10x11' operation = ImmediateOp_BIC;
    when '110x0' operation = ImmediateOp_MOVI;
    when '110x1' operation = ImmediateOp_MVNI;
    when '1110x' operation = ImmediateOp_MOVI;
    when '11110' operation = ImmediateOp_MOVI;
    when '11111'
        // FMOV Dn,#imm is in main FP instruction set
        if Q == '0' then UNDEFINED;
        operation = ImmediateOp_MOVI;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

Assembler Symbols

<Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<imm> Is a 64-bit immediate 'aaaaaaaabbbbbbbbccccccccddddddddeeeeeeeeffffffffgggggggghhhhhhhh', encoded in "a:b:c:d:e:f:g:h".

<T> For the 8-bit variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

For the 16-bit variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the 32-bit variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	2S
1	4S

<imm8> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

<amount> For the 16-bit shifted immediate variant: is the shift amount encoded in “cmode<1>”:

cmode<1>	<amount>
0	0
1	8

defaulting to 0 if LSL is omitted.

For the 32-bit shifted immediate variant: is the shift amount encoded in “cmode<2:1>”:

cmode<2:1>	<amount>
00	0
01	8
10	16
11	24

defaulting to 0 if LSL is omitted.

For the 32-bit shifting ones variant: is the shift amount encoded in “cmode<0>”:

cmode<0>	<amount>
0	8
1	16

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
    when ImmediateOp_MOVI
        result = imm;
    when ImmediateOp_MVNI
        result = NOT(imm);
    when ImmediateOp_ORR
        operand = V[rd, datasize];
        result = operand OR imm;
    when ImmediateOp_BIC
        operand = V[rd, datasize];
        result = operand AND NOT(imm);

V[rd, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

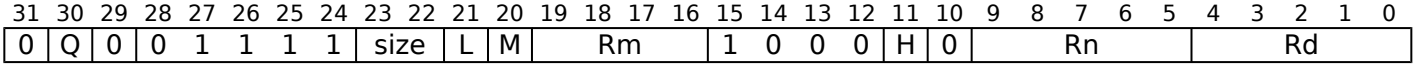
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MUL (by element)

Multiply (vector, by element). This instruction multiplies the vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



MUL <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) product;

element2 = UInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = UInt(Elem[operand1, e, esize]);
    product = (element1*element2)<esize-1:0>;
    Elem[result, e, esize] = product;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

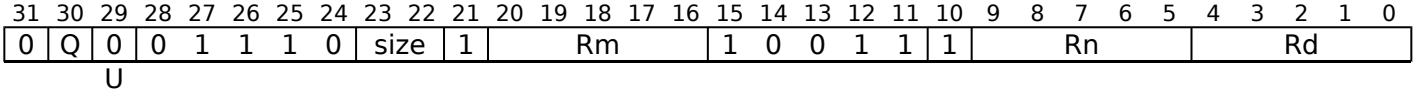
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MUL (vector)

Multiply (vector). This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



MUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if U == '1' && size != '00' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean poly = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if poly then
        product = PolynomialMult(element1, element2)<esize-1:0>;
    else
        product = (UInt(element1)*UInt(element2))<esize-1:0>;
    Elem[result, e, esize] = product;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

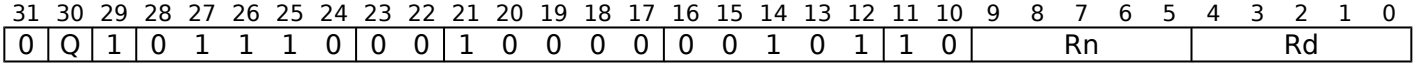
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MVN

Bitwise NOT (vector). This instruction reads each vector element from the source SIMD&FP register, places the inverse of each value into a vector, and writes the vector to the destination SIMD&FP register.
Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of NOT. This means:

- The encodings in this description are named to match the encodings of NOT.
- The description of NOT gives the operational pseudocode for this instruction.



MVN <Vd>.<T>, <Vn>.<T>

is equivalent to

NOT <Vd>.<T>, <Vn>.<T>

and is always the preferred disassembly.

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

The description of NOT gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

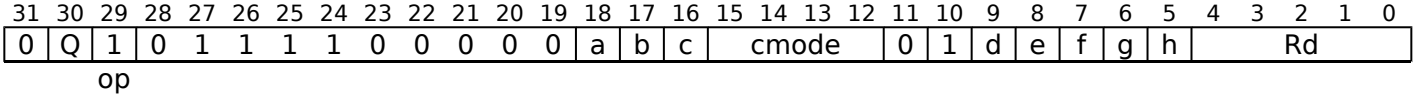
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MVNI

Move inverted Immediate (vector). This instruction places the inverse of an immediate constant into every vector element of the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



16-bit shifted immediate (cmode == 10x0)

```
MVNI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

32-bit shifted immediate (cmode == 0xx0)

```
MVNI <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

32-bit shifting ones (cmode == 110x)

```
MVNI <Vd>.<T>, #<imm8>, MSL #<amount>
```

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
    when '0xx01' operation = ImmediateOp_MVNI;
    when '0xx11' operation = ImmediateOp_BIC;
    when '10x01' operation = ImmediateOp_MVNI;
    when '10x11' operation = ImmediateOp_BIC;
    when '110x1' operation = ImmediateOp_MVNI;
    when '1110x' operation = ImmediateOp_MOVI;
    when '11111'
        // FMOV Dn,#imm is in main FP instruction set
        if Q == '0' then UNDEFINED;
        operation = ImmediateOp_MOVI;

imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the 16-bit variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the 32-bit variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	2S
1	4S

<imm8> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

<amount> For the 16-bit shifted immediate variant: is the shift amount encoded in “cmode<1>”:

cmode<1>	<amount>
0	0
1	8

defaulting to 0 if LSL is omitted.

For the 32-bit shifted immediate variant: is the shift amount encoded in “cmode<2:1>”:

cmode<2:1>	<amount>
00	0
01	8
10	16
11	24

defaulting to 0 if LSL is omitted.

For the 32-bit shifting ones variant: is the shift amount encoded in “cmode<0>”:

cmode<0>	<amount>
0	8
1	16

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
    when ImmediateOp_MOVI
        result = imm;
    when ImmediateOp_MVNI
        result = NOT(imm);
    when ImmediateOp_ORR
        operand = V[rd, datasize];
        result = operand OR imm;
    when ImmediateOp_BIC
        operand = V[rd, datasize];
        result = operand AND NOT(imm);

V[rd, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

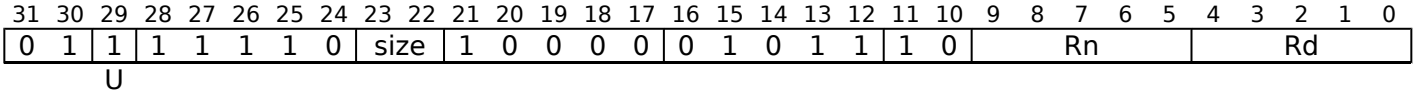
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

NEG (vector)

Negate (vector). This instruction reads each vector element from the source SIMD&FP register, negates each value, puts the result into a vector, and writes the vector to the destination SIMD&FP register.
Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.
It has encodings from 2 classes: Scalar and Vector

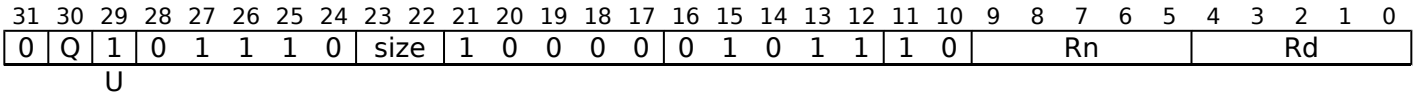
Scalar



NEG <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');
```

Vector



NEG <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

NOT

Bitwise NOT (vector). This instruction reads each vector element from the source SIMD&FP register, places the inverse of each value into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MVN.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	0	0	1	0	0	0	0	0	0	1	0	1	1	0	Rn					Rd				

NOT <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = NOT(element);

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

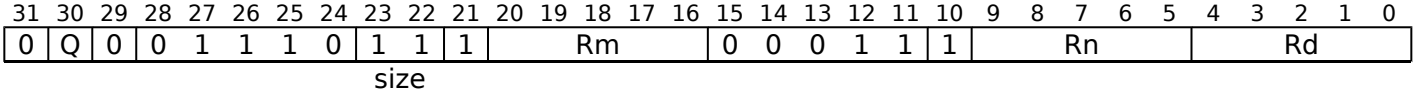
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ORN (vector)

Bitwise inclusive OR NOT (vector). This instruction performs a bitwise OR NOT between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



ORN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

operand2 = NOT(operand2);

result = operand1 OR operand2;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

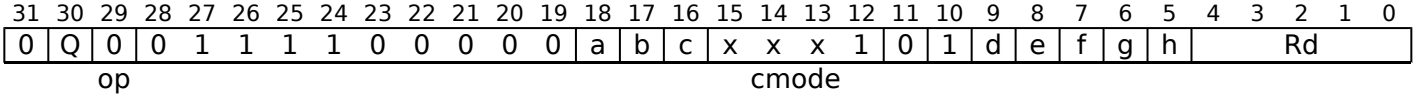
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ORR (vector, immediate)

Bitwise inclusive OR (vector, immediate). This instruction reads each vector element from the destination SIMD&FP register, performs a bitwise OR between each result and an immediate constant, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



16-bit (cmode == 10x1)

```
ORR <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

32-bit (cmode == 0xx1)

```
ORR <Vd>.<T>, #<imm8>{, LSL #<amount>}
```

```
integer rd = UInt(Rd);

integer datasize = if Q == '1' then 128 else 64;
bits(datasize) imm;
bits(64) imm64;

ImmediateOp operation;
case cmode:op of
    when '0xx00' operation = ImmediateOp_MOVI;
    when '0xx10' operation = ImmediateOp_ORR;
    when '10x00' operation = ImmediateOp_MOVI;
    when '10x10' operation = ImmediateOp_ORR;
    when '110x0' operation = ImmediateOp_MOVI;
    when '1110x' operation = ImmediateOp_MOVI;
    when '11110' operation = ImmediateOp_MOVI;
imm64 = AdvSIMDExpandImm(op, cmode, a:b:c:d:e:f:g:h);
imm = Replicate(imm64, datasize DIV 64);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP register, encoded in the "Rd" field.

<T> For the 16-bit variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the 32-bit variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	2S
1	4S

<imm8> Is an 8-bit immediate encoded in "a:b:c:d:e:f:g:h".

<amount> For the 16-bit variant: is the shift amount encoded in “cmode<1>”:

cmode<1>	<amount>
0	0
1	8

defaulting to 0 if LSL is omitted.

For the 32-bit variant: is the shift amount encoded in “cmode<2:1>”:

cmode<2:1>	<amount>
00	0
01	8
10	16
11	24

defaulting to 0 if LSL is omitted.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand;
bits(datasize) result;

case operation of
    when ImmediateOp_MOVI
        result = imm;
    when ImmediateOp_MVNI
        result = NOT(imm);
    when ImmediateOp_ORR
        operand = V[rd, datasize];
        result = operand OR imm;
    when ImmediateOp_BIC
        operand = V[rd, datasize];
        result = operand AND NOT(imm);

V[rd, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

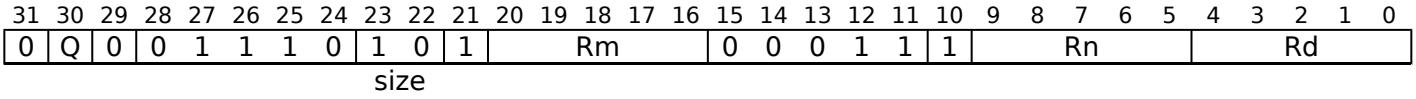
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ORR (vector, register)

Bitwise inclusive OR (vector, register). This instruction performs a bitwise OR between the two source SIMD&FP registers, and writes the result to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (vector).




```
ORR <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer datasize = if Q == '1' then 128 else 64;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Alias Conditions

Alias	Is preferred when
MOV (vector)	`Rm == Rn`

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

result = operand1 OR operand2;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

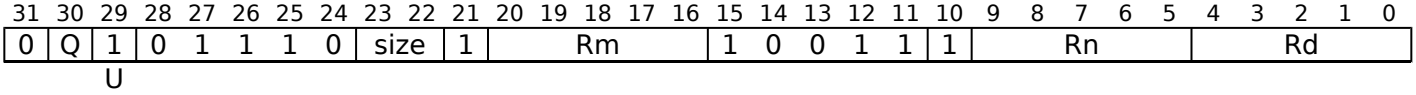
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PMUL

Polynomial Multiply. This instruction multiplies corresponding elements in the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

For information about multiplying polynomials see *Polynomial arithmetic over {0, 1}*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



PMUL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if U == '1' && size != '00' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean poly = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	x	RESERVED
1x	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;
bits(esize) product;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if poly then
        product = PolynomialMult(element1, element2)<esize-1:0>;
    else
        product = (UInt(element1)*UInt(element2))<esize-1:0>;
    Elem[result, e, esize] = product;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PMULL, PMULL2

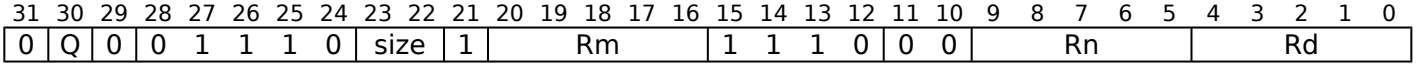
Polynomial Multiply Long. This instruction multiplies corresponding elements in the lower or upper half of the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

For information about multiplying polynomials, see *Polynomial arithmetic over {0, 1}*.

The PMULL instruction extracts each source vector from the lower half of each source register. The PMULL2 instruction extracts each source vector from the upper half of each source register.

The PMULL, PMULL2 variants that operate on 64-bit data quantities are defined only when FEAT_PMULL is implemented.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



PMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '01' || size == '10' then UNDEFINED;
if size == '11' && !HaveBit128PMULLExt() then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	RESERVED
10	RESERVED
11	1Q

The '1Q' arrangement is only allocated in an implementation that includes the Cryptographic Extension, and is otherwise RESERVED.

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	x	RESERVED
10	x	RESERVED
11	0	1D
11	1	2D

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    Elem[result, e, 2*esize] = PolynomialMult(element1, element2);

V[d, 2*datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

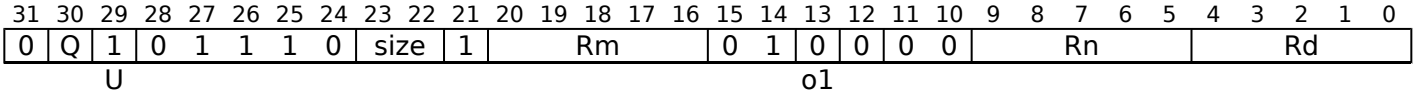
RADDHN, RADDHN2

Rounding Add returning High Narrow. This instruction adds each vector element in the first source SIMD&FP register to the corresponding vector element in the second source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

The results are rounded. For truncated results, see *ADDHN*.

The RADDHN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RADDHN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



RADDHN{2} <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n, 2*datasize];
bits(2*datasize) operand2 = V[m, 2*datasize];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;

for e = 0 to elements-1
    element1 = Elem[operand1, e, 2*esize];
    element2 = Elem[operand2, e, 2*esize];
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    sum = sum + round_const;
    Elem[result, e, esize] = sum<2*esize-1:esize>;

Vpart[d, part, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

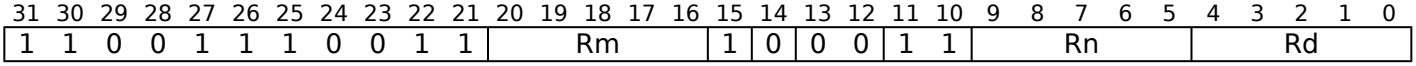
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

RAX1

Rotate and Exclusive OR rotates each 64-bit element of the 128-bit vector in a source SIMD&FP register left by 1, performs a bitwise exclusive OR of the resulting 128-bit vector and the vector in another source SIMD&FP register, and writes the result to the destination SIMD&FP register.

This instruction is implemented only when *FEAT_SHA3* is implemented.

Advanced SIMD
(FEAT_SHA3)



RAX1 <Vd>.2D, <Vn>.2D, <Vm>.2D

```
if !HaveSHA3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn>	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
V[d, 128] = Vn EOR (ROL(Vm<127:64>, 1):ROL(Vm<63:0>, 1));
```

Operational information

If PSTATE.DIT is 1:

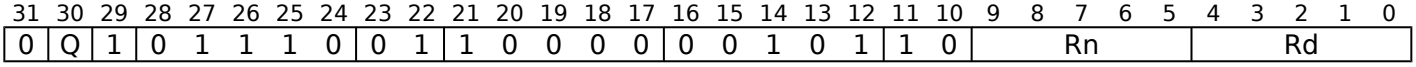
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

RBIT (vector)

Reverse Bit order (vector). This instruction reads each vector element from the source SIMD&FP register, reverses the bits of the element, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



RBIT <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “Q”:

Q	<T>
0	8B
1	16B

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(esize) element;
bits(esize) rev;

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    for i = 0 to esize-1
        rev<(esize-1)-i> = element<i>;
    Elem[result, e, esize] = rev;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

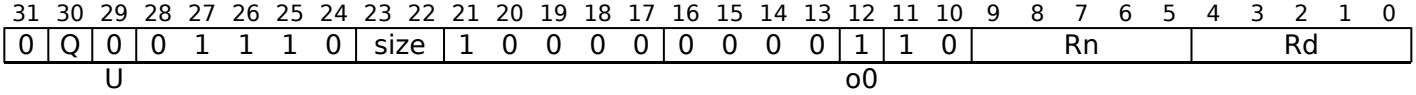
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

REV16 (vector)

Reverse elements in 16-bit halfwords (vector). This instruction reverses the order of 8-bit elements in each halfword of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



REV16 <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

// size=esize:   B(0),  H(1),  S(1), D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx: 64(0), 32(1), 16(2)
bits(2) op = o0:U;

// => op+size:
//    64+B = 0, 64+H = 1, 64+S = 2, 64+D = X
//    32+B = 1, 32+H = 2, 32+S = X, 32+D = X
//    16+B = 2, 16+H = X, 16+S = X, 16+D = X
//     8+B = X,  8+H = X,  8+S = X,  8+D = X
// => 3-(op+size) (index bits in group)
//    64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
//    32+B = 2, 32+H = 1, 32+S = X, 32+D = X
//    16+B = 1, 16+H = X, 16+S = X, 16+D = X
//     8+B = X,  8+H = X,  8+S = X,  8+D = X

// index bits within group: 1, 2, 3
if UInt(op) + UInt(size) >= 3 then UNDEFINED;

integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	x	RESERVED
1x	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element = 0;
integer rev_element;
for c = 0 to containers-1
    rev_element = element + elements_per_container - 1;
    for e = 0 to elements_per_container-1
        Elem[result, rev_element, esize] = Elem[operand, element, esize];
        element = element + 1;
        rev_element = rev_element - 1;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

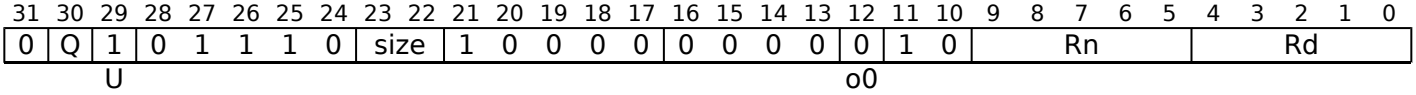
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

REV32 (vector)

Reverse elements in 32-bit words (vector). This instruction reverses the order of 8-bit or 16-bit elements in each word of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



REV32 <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

// size=esize:   B(0),  H(1),  S(1), D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx: 64(0), 32(1), 16(2)
bits(2) op = o0:U;

// => op+size:
//    64+B = 0, 64+H = 1, 64+S = 2, 64+D = X
//    32+B = 1, 32+H = 2, 32+S = X, 32+D = X
//    16+B = 2, 16+H = X, 16+S = X, 16+D = X
//     8+B = X,  8+H = X,  8+S = X,  8+D = X
// => 3-(op+size) (index bits in group)
//    64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
//    32+B = 2, 32+H = 1, 32+S = X, 32+D = X
//    16+B = 1, 16+H = X, 16+S = X, 16+D = X
//     8+B = X,  8+H = X,  8+S = X,  8+D = X

// index bits within group: 1, 2, 3
if UInt(op) + UInt(size) >= 3 then UNDEFINED;

integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
1x	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element = 0;
integer rev_element;
for c = 0 to containers-1
    rev_element = element + elements_per_container - 1;
    for e = 0 to elements_per_container-1
        Elem[result, rev_element, esize] = Elem[operand, element, esize];
        element = element + 1;
        rev_element = rev_element - 1;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

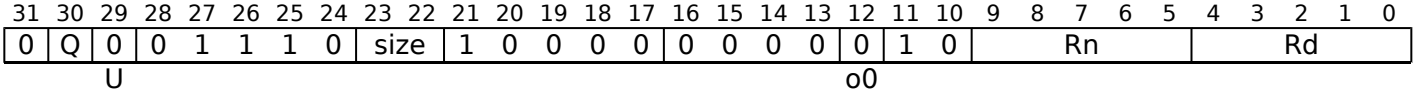
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

REV64

Reverse elements in 64-bit doublewords (vector). This instruction reverses the order of 8-bit, 16-bit, or 32-bit elements in each doubleword of the vector in the source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



REV64 <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

// size=esize:   B(0),  H(1),  S(1), D(S)
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;

// op=REVx: 64(0), 32(1), 16(2)
bits(2) op = o0:U;

// => op+size:
//    64+B = 0, 64+H = 1, 64+S = 2, 64+D = X
//    32+B = 1, 32+H = 2, 32+S = X, 32+D = X
//    16+B = 2, 16+H = X, 16+S = X, 16+D = X
//     8+B = X,  8+H = X,  8+S = X,  8+D = X
// => 3-(op+size) (index bits in group)
//    64/B = 3, 64+H = 2, 64+S = 1, 64+D = X
//    32+B = 2, 32+H = 1, 32+S = X, 32+D = X
//    16+B = 1, 16+H = X, 16+S = X, 16+D = X
//     8+B = X,  8+H = X,  8+S = X,  8+D = X

// index bits within group: 1, 2, 3
if UInt(op) + UInt(size) >= 3 then UNDEFINED;

integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;

integer containers = datasize DIV container_size;
integer elements_per_container = container_size DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element = 0;
integer rev_element;
for c = 0 to containers-1
    rev_element = element + elements_per_container - 1;
    for e = 0 to elements_per_container-1
        Elem[result, rev_element, esize] = Elem[operand, element, esize];
        element = element + 1;
        rev_element = rev_element - 1;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

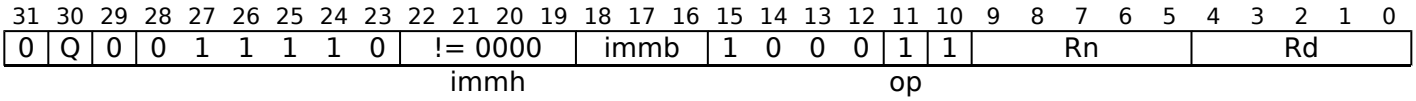
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

RSHRN, RSHRN2

Rounding Shift Right Narrow (immediate). This instruction reads each unsigned integer value from the vector in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The results are rounded. For truncated results, see *SHRN*.

The RSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



RSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<shift> Is the right shift amount, in the range 1 to the destination element width in bits, encoded in "immh:immb":

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n, datasize*2];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

for e = 0 to elements-1
    element = (UInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
    Elem[result, e, esize] = element<esize-1:0>;

Vpart[d, part, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

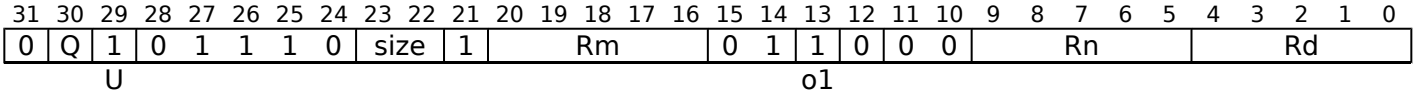
RSUBHN, RSUBHN2

Rounding Subtract returning High Narrow. This instruction subtracts each vector element of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register.

The results are rounded. For truncated results, see *SUBHN*.

The RSUBHN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RSUBHN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



RSUBHN{2} <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n, 2*datasize];
bits(2*datasize) operand2 = V[m, 2*datasize];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;

for e = 0 to elements-1
    element1 = Elem[operand1, e, 2*esize];
    element2 = Elem[operand2, e, 2*esize];
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    sum = sum + round_const;
    Elem[result, e, esize] = sum<2*esize-1:esize>;

Vpart[d, part, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

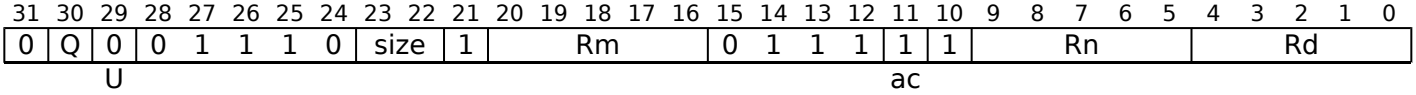
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SABA

Signed Absolute difference and Accumulate. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the elements of the vector of the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SABA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

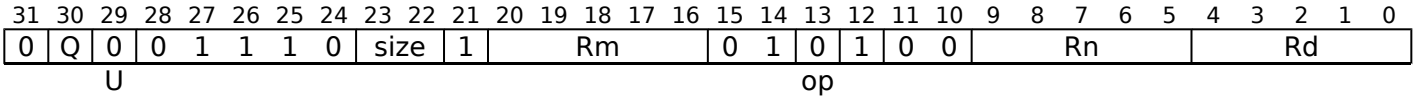
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SABAL, SABAL2

Signed Absolute difference and Accumulate Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The SABAL instruction extracts each source vector from the lower half of each source register. The SABAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SABAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d, 2*datasize] else Zeros(2 * datasize);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d, 2*datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

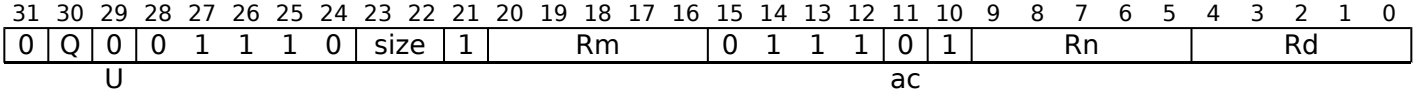
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SABD

Signed Absolute Difference. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, places the absolute values of the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

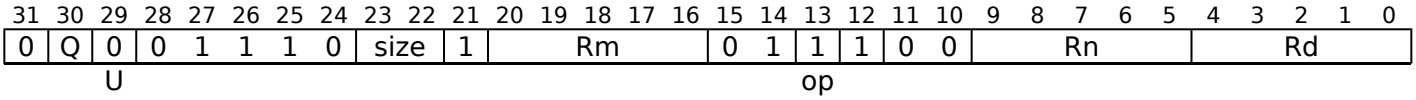
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SABDL, SABDL2

Signed Absolute Difference Long. This instruction subtracts the vector elements of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, places the absolute value of the results into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The SABDL instruction extracts each source vector from the lower half of each source register, while the SABDL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SABDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d, 2*datasize] else Zeros(2 * datasize);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d, 2*datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

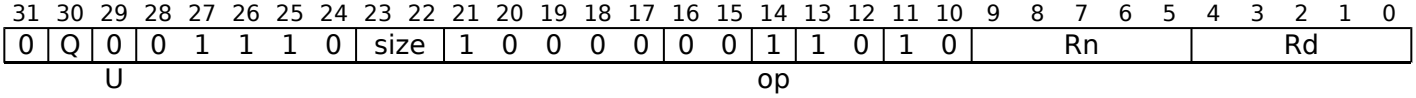
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SADALP

Signed Add and Accumulate Long Pairwise. This instruction adds pairs of adjacent signed integer values from the vector in the source SIMD&FP register and accumulates the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SADALP <Vd>.<Ta>, <Vn>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

if acc then result = V[d, datasize];
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>;
    if acc then
        Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
    else
        Elem[result, e, 2*esize] = sum;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

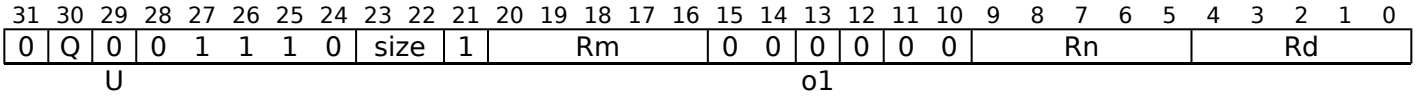
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SADDL, SADDL2

Signed Add Long (vector). This instruction adds each vector element in the lower or upper half of the first source SIMD&FP register to the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The SADDL instruction extracts each source vector from the lower half of each source register. The SADDL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SADDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d, 2*datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

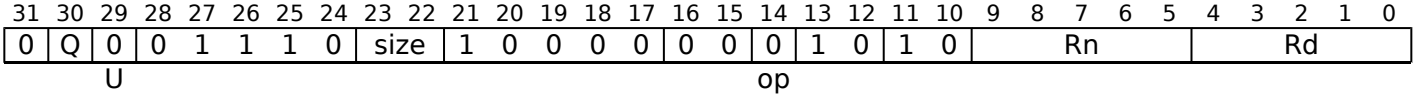
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SADDLP

Signed Add Long Pairwise. This instruction adds pairs of adjacent signed integer values from the vector in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SADDLP <Vd>.<Ta>, <Vn>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

if acc then result = V[d, datasize];
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>;
    if acc then
        Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
    else
        Elem[result, e, 2*esize] = sum;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

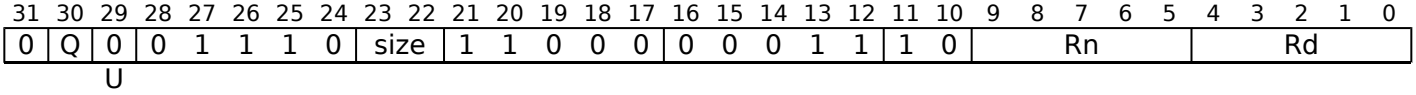
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SADDLV

Signed Add Long across Vector. This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register. The destination scalar is twice as long as the source vector elements. All the values in this instruction are signed integer values.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SADDLV <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

size	<V>
00	H
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
integer sum;

sum = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    sum = sum + Int(Elem[operand, e, esize], unsigned);

V[d, 2*esize] = sum<2*esize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

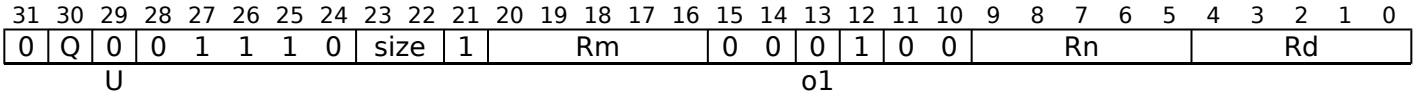
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SADDW, SADDW2

Signed Add Wide. This instruction adds vector elements of the first source SIMD&FP register to the corresponding vector elements in the lower or upper half of the second source SIMD&FP register, places the results in a vector, and writes the vector to the SIMD&FP destination register.

The SADDW instruction extracts the second source vector from the lower half of the second source register. The SADDW2 instruction extracts the second source vector from the upper half of the second source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SADDW{2} <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n, 2*datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, 2*esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d, 2*datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

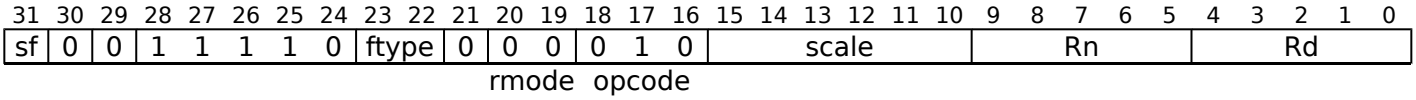
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SCVTF (scalar, fixed-point)

Signed fixed-point Convert to Floating-point (scalar). This instruction converts the signed value in the 32-bit or 64-bit general-purpose source register to a floating-point value using the rounding mode that is specified by the *FPCR*, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.



32-bit to half-precision (sf == 0 && ftype == 11)
(FEAT_FP16)

```
SCVTF <Hd>, <Wn>, #<fbits>
```

32-bit to single-precision (sf == 0 && ftype == 00)

```
SCVTF <Sd>, <Wn>, #<fbits>
```

32-bit to double-precision (sf == 0 && ftype == 01)

```
SCVTF <Dd>, <Wn>, #<fbits>
```

64-bit to half-precision (sf == 1 && ftype == 11)
(FEAT_FP16)

```
SCVTF <Hd>, <Xn>, #<fbits>
```

64-bit to single-precision (sf == 1 && ftype == 00)

```
SCVTF <Sd>, <Xn>, #<fbits>
```

64-bit to double-precision (sf == 1 && ftype == 01)

```
SCVTF <Dd>, <Xn>, #<fbits>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case ftype of
    when '00' fltsize = 32;
    when '01' fltsize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);

rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<fbits>	For the 32-bit to double-precision, 32-bit to half-precision and 32-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 32, encoded as 64 minus "scale". For the 64-bit to double-precision, 64-bit to half-precision and 64-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 64, encoded as 64 minus "scale".

Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
integer fsize = if merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;

intval = X[n, intsize];
fltval = if merge then V[d, fsize] else Zeros(fsize);
Elem[fltval, 0, fltsize] = FixedToFP(intval, fracbits, FALSE, fpcr, rounding, fltsize);
V[d, fsize] = fltval;
```

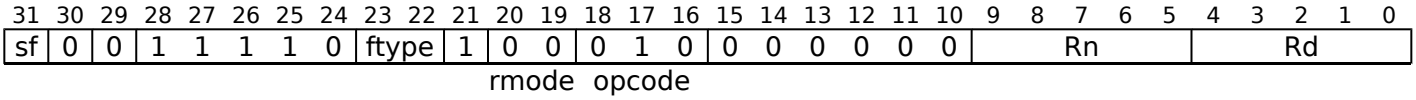
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SCVTF (scalar, integer)

Signed integer Convert to Floating-point (scalar). This instruction converts the signed integer value in the general-purpose source register to a floating-point value using the rounding mode that is specified by the *FPCR*, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



32-bit to half-precision (sf == 0 && ftype == 11)
(FEAT_FP16)

```
SCVTF <Hd>, <Wn>
```

32-bit to single-precision (sf == 0 && ftype == 00)

```
SCVTF <Sd>, <Wn>
```

32-bit to double-precision (sf == 0 && ftype == 01)

```
SCVTF <Dd>, <Wn>
```

64-bit to half-precision (sf == 1 && ftype == 11)
(FEAT_FP16)

```
SCVTF <Hd>, <Xn>
```

64-bit to single-precision (sf == 1 && ftype == 00)

```
SCVTF <Sd>, <Xn>
```

64-bit to double-precision (sf == 1 && ftype == 01)

```
SCVTF <Dd>, <Xn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
integer fsize = if merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;

intval = X[n, intsize];
fltval = if merge then V[d, fsize] else Zeros(fsize);
Elem[fltval, 0, fltsize] = FixedToFP(intval, 0, FALSE, fpcr, rounding, fltsize);
V[d, fsize] = fltval;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SCVTF (vector, fixed-point)

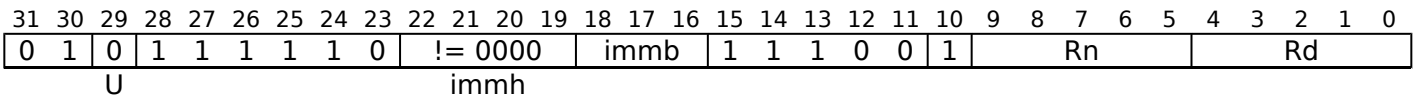
Signed fixed-point Convert to Floating-point (vector). This instruction converts each element in a vector from fixed-point to floating-point using the rounding mode that is specified by the *FPCR*, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

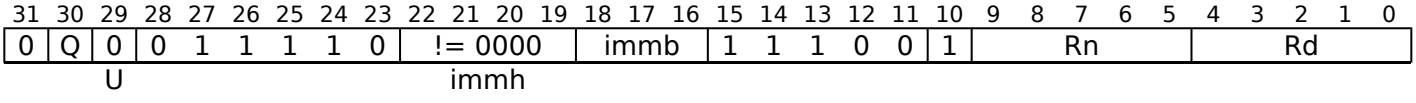
Scalar



SCVTF <V><d>, <V><n>, #<fbits>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh IN {'000x'} || (immh IN {'001x'} && !HaveFP16Ext()) then UNDEFINED;
integer esize = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR[]);
```

Vector



SCVTF <Vd>.<T>, <Vn>.<T>, #<fbits>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh IN {'000x'} || (immh IN {'001x'} && !HaveFP16Ext()) then UNDEFINED;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
000x	RESERVED
001x	H
01xx	S
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	x	RESERVED
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in “immh:immb”:

immh	<fbits>
000x	RESERVED
001x	(32-Uint(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in “immh:immb”:

immh	<fbits>
0000	SEE Advanced SIMD modified immediate
0001	RESERVED
001x	(32-Uint(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, fracbits, unsigned, fpcr, rounding, esize);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SCVTF (vector, integer)

Signed integer Convert to Floating-point (vector). This instruction converts each element in a vector from signed integer to floating-point using the rounding mode that is specified by the *FPCR*, and writes the result to the SIMD&FP destination register.

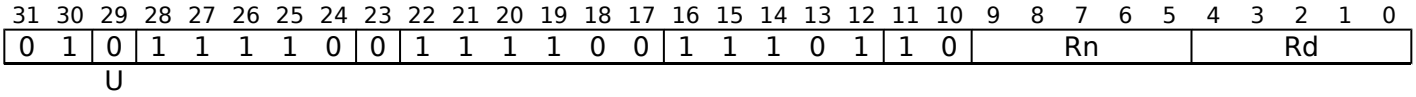
A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

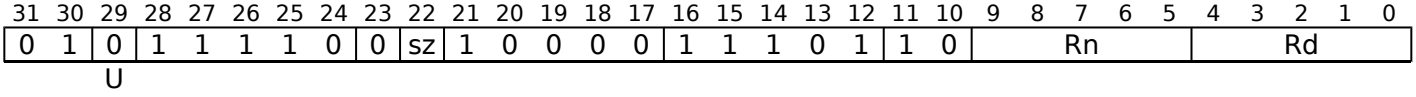
(FEAT_FP16)



SCVTF <Hd>, <Hn>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

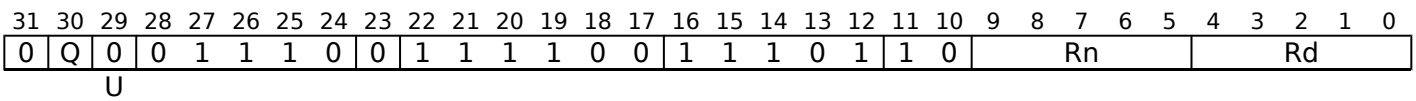


SCVTF <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Vector half precision

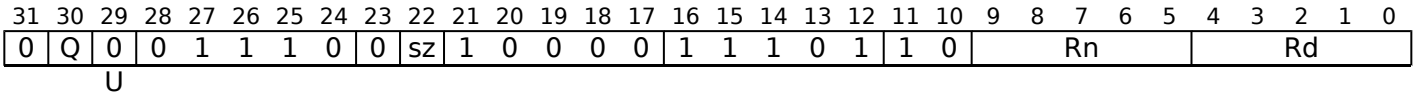
(FEAT_FP16)



SCVTF <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



SCVTF <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

FPRounding rounding = FPRoundingMode(fpcr);
bits(esize) element;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, 0, unsigned, fpcr, rounding, esize);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

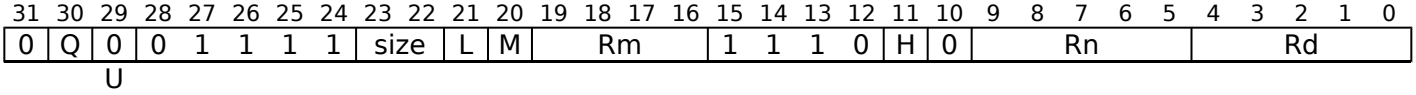
SDOT (by element)

Dot Product signed arithmetic (vector, by element). This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.
Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.
In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_AA64ISAR0_EL1.DP indicates whether this instruction is supported.

Vector
(FEAT_DotProd)



SDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.4B[<index>]

```
if !HaveDOTPExt() then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U == '0');

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer index = UInt(H:L);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	8B
1	16B

<Vm> Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<index> Is the element index, encoded in the "H:L" fields.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(128) operand2 = V[m, 128];
bits(datasize) result = V[d, datasize];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        if signed then
            element1 = SInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4*index+i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4*index+i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SDOT (vector)

Dot Product signed arithmetic (vector). This instruction performs the dot product of the four signed 8-bit elements in each 32-bit element of the first source register with the four signed 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

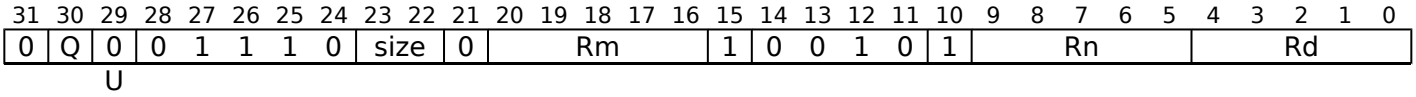
Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_AA64ISAR0_EL1.DP indicates whether this instruction is supported.

Vector
(FEAT_DotProd)



SDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
if !HaveDOTPExt() then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U == '0');
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	8B
1	16B

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

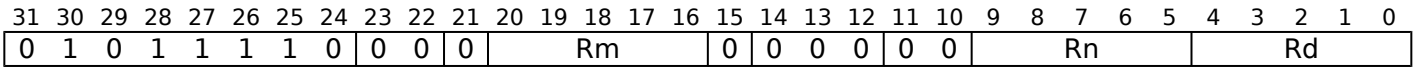
```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

result = V[d, datasize];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        if signed then
            element1 = SInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4*e+i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4*e+i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SHA1C

SHA1 hash update (choose).

Advanced SIMD
(FEAT_SHA1)



SHA1C <Qd>, <Sn>, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;
```

Assembler Symbols

<Qd>	Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) x = V[d, 128];
bits(32) y = V[n, 32];    // Note: 32 not 128 bits wide
bits(128) w = V[m, 128];
bits(32) t;

for e = 0 to 3
    t = SHAchoose(x<63:32>, x<95:64>, x<127:96>);
    y = y + ROL(x<31:0>, 5) + t + Elem[w, e, 32];
    x<63:32> = ROL(x<63:32>, 30);
    <y, x> = ROL(y:x, 32);
V[d, 128] = x;
```

Operational information

If PSTATE.DIT is 1:

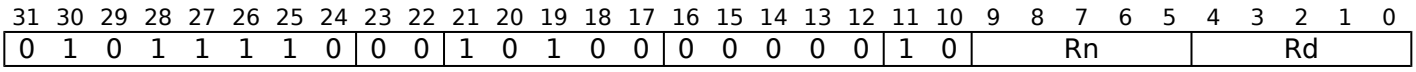
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SHA1H

SHA1 fixed rotate.

Advanced SIMD
(FEAT_SHA1)



SHA1H <Sd>, <Sn>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveSHA1Ext() then UNDEFINED;
```

Assembler Symbols

<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(32) operand = V[n, 32];    // read element [0] only,  [1-3] zeroed
V[d, 32] = ROL(operand, 30);
```

Operational information

If PSTATE.DIT is 1:

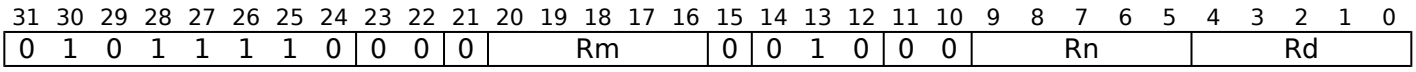
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SHA1M

SHA1 hash update (majority).

Advanced SIMD
(FEAT_SHA1)



SHA1M <Qd>, <Sn>, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;
```

Assembler Symbols

<Qd>	Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) x = V[d, 128];
bits(32) y = V[n, 32];    // Note: 32 not 128 bits wide
bits(128) w = V[m, 128];
bits(32) t;

for e = 0 to 3
    t = SHAmajority(x<63:32>, x<95:64>, x<127:96>);
    y = y + ROL(x<31:0>, 5) + t + Elem[w, e, 32];
    x<63:32> = ROL(x<63:32>, 30);
    <y, x> = ROL(y:x, 32);
V[d, 128] = x;
```

Operational information

If PSTATE.DIT is 1:

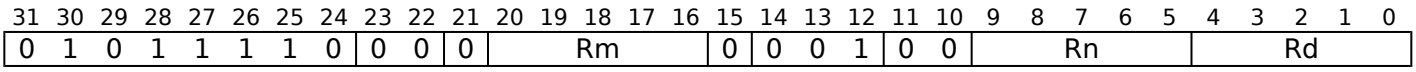
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SHA1P

SHA1 hash update (parity).

Advanced SIMD
(FEAT_SHA1)



SHA1P <Qd>, <Sn>, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;
```

Assembler Symbols

<Qd>	Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
<Sn>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) x = V[d, 128];
bits(32) y = V[n, 32];    // Note: 32 not 128 bits wide
bits(128) w = V[m, 128];
bits(32) t;

for e = 0 to 3
    t = SHAparity(x<63:32>, x<95:64>, x<127:96>);
    y = y + ROL(x<31:0>, 5) + t + Elem[w, e, 32];
    x<63:32> = ROL(x<63:32>, 30);
    <y, x> = ROL(y:x, 32);
V[d, 128] = x;
```

Operational information

If PSTATE.DIT is 1:

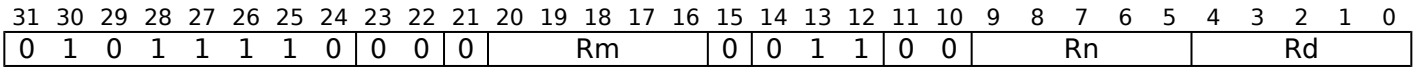
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SHA1SU0

SHA1 schedule update 0.

Advanced SIMD
(FEAT_SHA1)



SHA1SU0 <Vd>.4S, <Vn>.4S, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA1Ext() then UNDEFINED;
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d, 128];
bits(128) operand2 = V[n, 128];
bits(128) operand3 = V[m, 128];
bits(128) result;

result = operand2<63:0>:operand1<127:64>;
result = result EOR operand1 EOR operand3;
V[d, 128] = result;
```

Operational information

If PSTATE.DIT is 1:

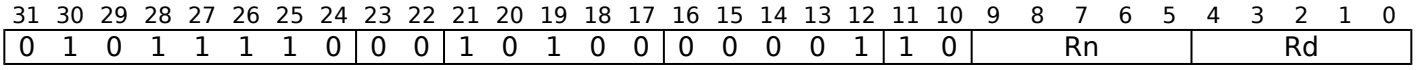
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SHA1SU1

SHA1 schedule update 1.

Advanced SIMD
(FEAT_SHA1)



SHA1SU1 <Vd>.4S, <Vn>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveSHA1Ext() then UNDEFINED;
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d, 128];
bits(128) operand2 = V[n, 128];
bits(128) result;
bits(128) T = operand1 EOR LSR(operand2, 32);
result<31:0> = ROL(T<31:0>, 1);
result<63:32> = ROL(T<63:32>, 1);
result<95:64> = ROL(T<95:64>, 1);
result<127:96> = ROL(T<127:96>, 1) EOR ROL(T<31:0>, 2);
V[d, 128] = result;
```

Operational information

If PSTATE.DIT is 1:

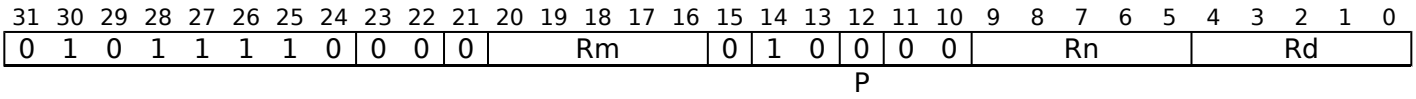
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SHA256H

SHA256 hash update (part 1).

Advanced SIMD
(FEAT_SHA256)



SHA256H <Qd>, <Qn>, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA256Ext() then UNDEFINED;
```

Assembler Symbols

<Qd>	Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
<Qn>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) result;
result = SHA256hash(V[d, 128], V[n, 128], V[m, 128], TRUE);
V[d, 128] = result;
```

Operational information

If PSTATE.DIT is 1:

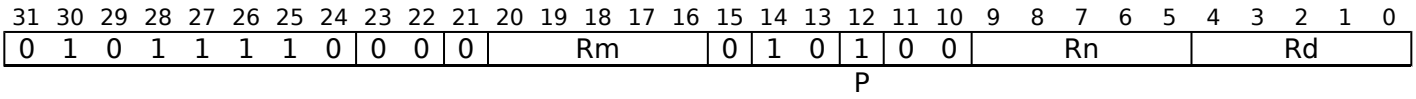
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SHA256H2

SHA256 hash update (part 2).

Advanced SIMD
(FEAT_SHA256)



SHA256H2 <Qd>, <Qn>, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA256Ext() then UNDEFINED;
```

Assembler Symbols

<Qd>	Is the 128-bit name of the SIMD&FP source and destination, encoded in the "Rd" field.
<Qn>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) result;
result = SHA256hash(V[n, 128], V[d, 128], V[m, 128], FALSE);
V[d, 128] = result;
```

Operational information

If PSTATE.DIT is 1:

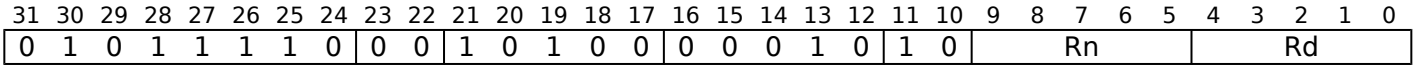
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SHA256SU0

SHA256 schedule update 0.

Advanced SIMD
(FEAT_SHA256)



SHA256SU0 <Vd>.4S, <Vn>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
if !HaveSHA256Ext() then UNDEFINED;
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d, 128];
bits(128) operand2 = V[n, 128];
bits(128) result;
bits(128) T = operand2<31:0>:operand1<127:32>;
bits(32) elt;

for e = 0 to 3
    elt = Elem[T, e, 32];
    elt = ROR(elt, 7) EOR ROR(elt, 18) EOR LSR(elt, 3);
    Elem[result, e, 32] = elt + Elem[operand1, e, 32];
V[d, 128] = result;
```

Operational information

If PSTATE.DIT is 1:

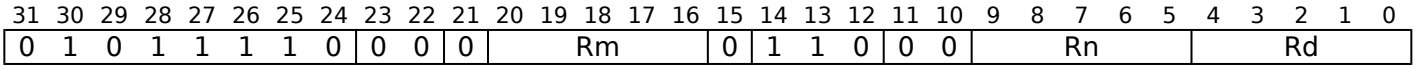
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SHA256SU1

SHA256 schedule update 1.

Advanced SIMD
(FEAT_SHA256)



SHA256SU1 <Vd>.4S, <Vn>.4S, <Vm>.4S

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if !HaveSHA256Ext() then UNDEFINED;
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) operand1 = V[d, 128];
bits(128) operand2 = V[n, 128];
bits(128) operand3 = V[m, 128];
bits(128) result;
bits(128) T0 = operand3<31:0>:operand2<127:32>;
bits(64) T1;
bits(32) elt;

T1 = operand3<127:64>;
for e = 0 to 1
    elt = Elem[T1, e, 32];
    elt = ROR(elt, 17) EOR ROR(elt, 19) EOR LSR(elt, 10);
    elt = elt + Elem[operand1, e, 32] + Elem[T0, e, 32];
    Elem[result, e, 32] = elt;

T1 = result<63:0>;
for e = 2 to 3
    elt = Elem[T1, e-2, 32];
    elt = ROR(elt, 17) EOR ROR(elt, 19) EOR LSR(elt, 10);
    elt = elt + Elem[operand1, e, 32] + Elem[T0, e, 32];
    Elem[result, e, 32] = elt;

V[d, 128] = result;
```

Operational information

If PSTATE.DIT is 1:

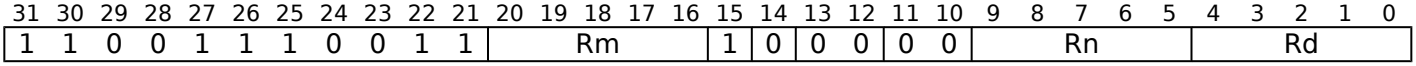
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

SHA512H

SHA512 Hash update part 1 takes the values from the three 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the sigma1 and chi functions of two iterations of the SHA512 computation. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when *FEAT_SHA512* is implemented.

Advanced SIMD
(FEAT_SHA512)



SHA512H <Qd>, <Qn>, <Vm>.2D

```
if !HaveSHA512Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

<Qd>	Is the 128-bit name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Qn>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vtmp;
bits(64) MSigma1;
bits(64) tmp;
bits(128) x = V[n, 128];
bits(128) y = V[m, 128];
bits(128) w = V[d, 128];

MSigma1 = ROR(y<127:64>, 14) EOR ROR(y<127:64>, 18) EOR ROR(y<127:64>, 41);
Vtmp<127:64> = (y<127:64> AND x<63:0>) EOR (NOT(y<127:64>) AND x<127:64>);
Vtmp<127:64> = (Vtmp<127:64> + MSigma1 + w<127:64>);
tmp = Vtmp<127:64> + y<63:0>;
MSigma1 = ROR(tmp, 14) EOR ROR(tmp, 18) EOR ROR(tmp, 41);
Vtmp<63:0> = (tmp AND y<127:64>) EOR (NOT(tmp) AND x<63:0>);
Vtmp<63:0> = (Vtmp<63:0> + MSigma1 + w<63:0>);
V[d, 128] = Vtmp;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

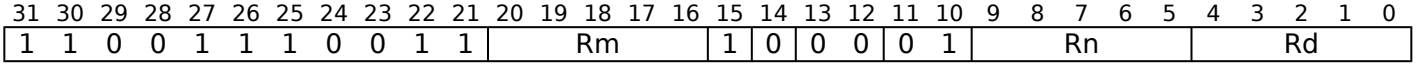
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SHA512H2

SHA512 Hash update part 2 takes the values from the three 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the sigma0 and majority functions of two iterations of the SHA512 computation. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when *FEAT_SHA512* is implemented.

Advanced SIMD
(FEAT_SHA512)



SHA512H2 <Qd>, <Qn>, <Vm>.2D

```
if !HaveSHA512Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

<Qd>	Is the 128-bit name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Qn>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vtmp;
bits(64) NSigma0;
bits(128) x = V[n, 128];
bits(128) y = V[m, 128];
bits(128) w = V[d, 128];

NSigma0 = ROR(y<63:0>, 28) EOR ROR(y<63:0>, 34) EOR ROR(y<63:0>, 39);
Vtmp<127:64> = (x<63:0> AND y<127:64>) EOR (x<63:0> AND y<63:0>) EOR (y<127:64> AND y<63:0>);
Vtmp<127:64> = (Vtmp<127:64> + NSigma0 + w<127:64>);
NSigma0 = ROR(Vtmp<127:64>, 28) EOR ROR(Vtmp<127:64>, 34) EOR ROR(Vtmp<127:64>, 39);
Vtmp<63:0> = (Vtmp<127:64> AND y<63:0>) EOR (Vtmp<127:64> AND y<127:64>) EOR (y<127:64> AND y<63:0>);
Vtmp<63:0> = (Vtmp<63:0> + NSigma0 + w<63:0>);

V[d, 128] = Vtmp;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

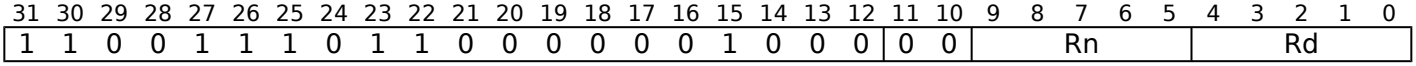
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SHA512SU0

SHA512 Schedule Update 0 takes the values from the two 128-bit source SIMD&FP registers and produces a 128-bit output value that combines the gamma0 functions of two iterations of the SHA512 schedule update that are performed after the first 16 iterations within a block. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when *FEAT_SHA512* is implemented.

Advanced SIMD
(FEAT_SHA512)



SHA512SU0 <Vd>.2D, <Vn>.2D

```
if !HaveSHA512Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(64) sig0;
bits(128) Vtmp;
bits(128) x = V[n, 128];
bits(128) w = V[d, 128];
sig0 = ROR(w<127:64>, 1) EOR ROR(w<127:64>, 8) EOR ('0000000':w<127:71>);
Vtmp<63:0> = w<63:0> + sig0;
sig0 = ROR(x<63:0>, 1) EOR ROR(x<63:0>, 8) EOR ('0000000':x<63:7>);
Vtmp<127:64> = w<127:64> + sig0;
V[d, 128] = Vtmp;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

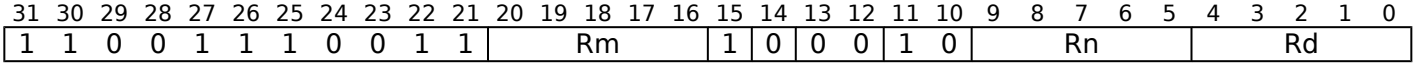
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SHA512SU1

SHA512 Schedule Update 1 takes the values from the three source SIMD&FP registers and produces a 128-bit output value that combines the gamma1 functions of two iterations of the SHA512 schedule update that are performed after the first 16 iterations within a block. It returns this value to the destination SIMD&FP register.

This instruction is implemented only when *FEAT_SHA512* is implemented.

Advanced SIMD
(FEAT_SHA512)



SHA512SU1 <Vd>.2D, <Vn>.2D, <Vm>.2D

```
if !HaveSHA512Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(64) sig1;
bits(128) Vtmp;
bits(128) x = V[n, 128];
bits(128) y = V[m, 128];
bits(128) w = V[d, 128];

sig1 = ROR(x<127:64>, 19) EOR ROR(x<127:64>, 61) EOR ('000000':x<127:70>);
Vtmp<127:64> = w<127:64> + sig1 + y<127:64>;
sig1 = ROR(x<63:0>, 19) EOR ROR(x<63:0>, 61) EOR ('000000':x<63:6>);
Vtmp<63:0> = w<63:0> + sig1 + y<63:0>;
V[d, 128] = Vtmp;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

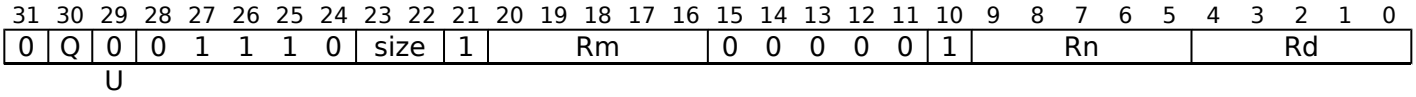
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SHADD

Signed Halving Add. This instruction adds corresponding signed integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see *SRHADD*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = (element1 + element2) >> 1;
    Elem[result, e, esize] = sum<esize-1:0>;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

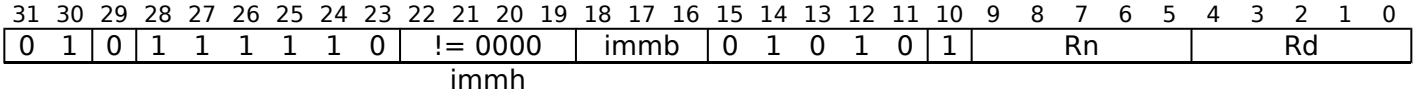
SHL

Shift Left (immediate). This instruction reads each value from a vector, left shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

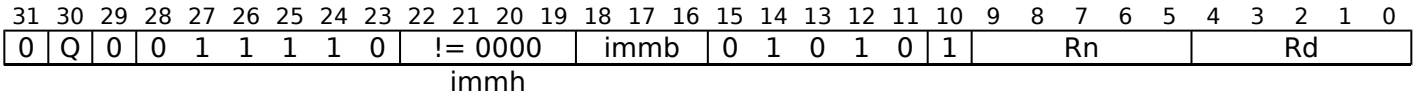
Scalar



SHL <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;
```

Vector



SHL <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to 63, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(UInt(immh:immb)-64)

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(UInt(immh:immb)-8)
001x	(UInt(immh:immb)-16)
01xx	(UInt(immh:immb)-32)
1xxx	(UInt(immh:immb)-64)

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;

for e = 0 to elements-1
    Elem[result, e, esize] = LSL(Elem[operand, e, esize], shift);

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

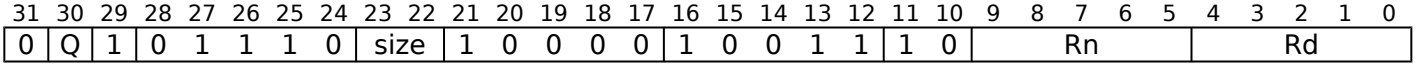
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SHLL, SHLL2

Shift Left Long (by element size). This instruction reads each vector element in the lower or upper half of the source SIMD&FP register, left shifts each result by the element size, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The SHLL instruction extracts vector elements from the lower half of the source register. The SHLL2 instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.




```
SHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #<shift>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = esize;
boolean unsigned = FALSE;    // Or TRUE without change of functionality
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<shift> Is the left shift amount, which must be equal to the source element width in bits, encoded in “size”:

size	<shift>
00	8
01	16
10	32
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part, datasize];
bits(2*datasize) result;
integer element;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], unsigned) << shift;
    Elem[result, e, 2*esize] = element<2*esize-1:0>;

V[d, 2*datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

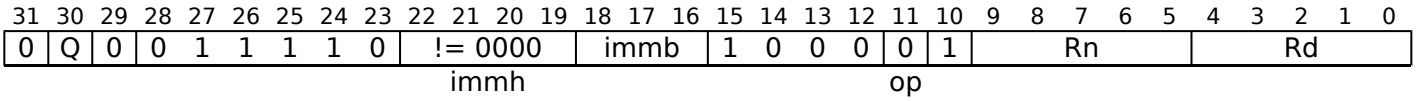
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SHRN, SHRN2

Shift Right Narrow (immediate). This instruction reads each unsigned integer value from the source SIMD&FP register, right shifts each result by an immediate value, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. The results are truncated. For rounded results, see *RSHRN*.

The RSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the RSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<shift> Is the right shift amount, in the range 1 to the destination element width in bits, encoded in "immh:immb":

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n, datasize*2];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

for e = 0 to elements-1
    element = (UInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
    Elem[result, e, esize] = element<esize-1:0>;

Vpart[d, part, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

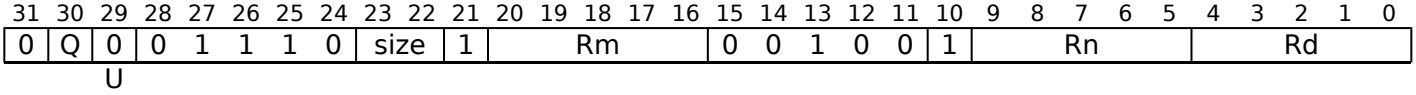
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SHSUB

Signed Halving Subtract. This instruction subtracts the elements in the vector in the second source SIMD&FP register from the corresponding elements in the vector in the first source SIMD&FP register, shifts each result right one bit, places each result into elements of a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SHSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer diff;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = (element1 - element2) >> 1;
    Elem[result, e, esize] = diff<esize-1:0>;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

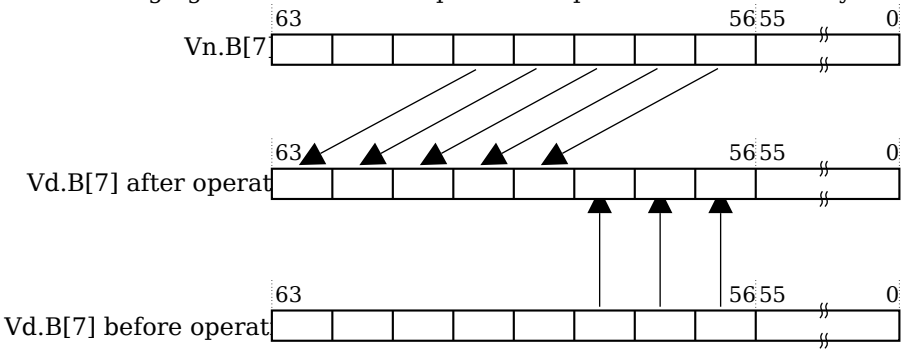
- The execution time of this instruction is independent of:

    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

SLI

Shift Left and Insert (immediate). This instruction reads each vector element in the source SIMD&FP register, left shifts each vector element by an immediate value, and inserts the result into the corresponding vector element in the destination SIMD&FP register such that the new zero bits created by the shift are not inserted but retain their existing value. Bits shifted out of the left of each vector element in the source register are lost.

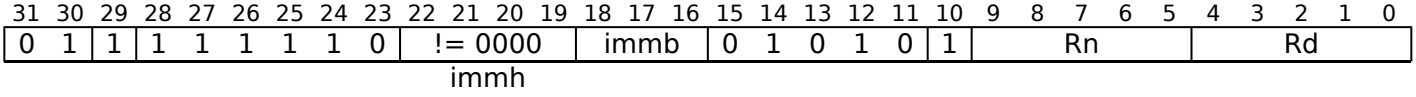
The following figure shows an example of the operation of shift left by 3 for an 8-bit vector element.



Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

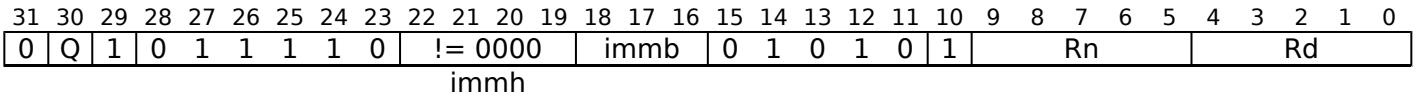
Scalar



SLI <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;
```

Vector



SLI <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to 63, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(UInt(immh:immb)-64)

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(UInt(immh:immb)-8)
001x	(UInt(immh:immb)-16)
01xx	(UInt(immh:immb)-32)
1xxx	(UInt(immh:immb)-64)

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) operand2 = V[d, datasize];
bits(datasize) result;
bits(esize) mask = LSL(Ones(esize), shift);
bits(esize) shifted;

for e = 0 to elements-1
    shifted = LSL(Elem[operand, e, esize], shift);
    Elem[result, e, esize] = (Elem[operand2, e, esize] AND NOT(mask)) OR shifted;
V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

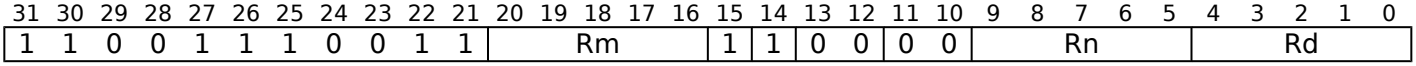
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

SM3PARTW1

SM3PARTW1 takes three 128-bit vectors from the three source SIMD&FP registers and returns a 128-bit result in the destination SIMD&FP register. The result is obtained by a three-way exclusive OR of the elements within the input vectors with some fixed rotations, see the Operation pseudocode for more information.

This instruction is implemented only when *FEAT_SM3* is implemented.

Advanced SIMD
(FEAT_SM3)



SM3PARTW1 <Vd>.4S, <Vn>.4S, <Vm>.4S

```
if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
bits(128) Vd = V[d, 128];
bits(128) result;

result<95:0> = (Vd EOR Vn)<95:0> EOR (ROL(Vm<127:96>, 15):ROL(Vm<95:64>, 15):ROL(Vm<63:32>, 15));

for i = 0 to 3
    if i == 3 then
        result<127:96> = (Vd EOR Vn)<127:96> EOR (ROL(result<31:0>, 15));
    result<(32*i)+31:(32*i)> = result<(32*i)+31:(32*i)> EOR ROL(result<(32*i)+31:(32*i)>, 15) EOR ROL(res
V[d, 128] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

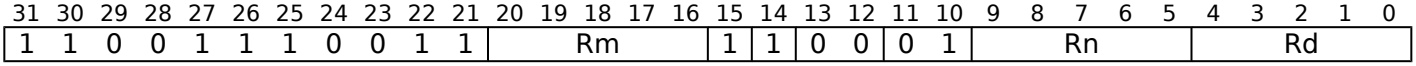
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SM3PARTW2

SM3PARTW2 takes three 128-bit vectors from three source SIMD&FP registers and returns a 128-bit result in the destination SIMD&FP register. The result is obtained by a three-way exclusive OR of the elements within the input vectors with some fixed rotations, see the Operation pseudocode for more information.

This instruction is implemented only when *FEAT_SM3* is implemented.

Advanced SIMD
(FEAT_SM3)



SM3PARTW2 <Vd>.4S, <Vn>.4S, <Vm>.4S

```
if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
bits(128) Vd = V[d, 128];
bits(128) result;
bits(128) tmp;
bits(32) tmp2;
tmp<127:0> = Vn EOR (ROL(Vm<127:96>, 7):ROL(Vm<95:64>, 7):ROL(Vm<63:32>, 7):ROL(Vm<31:0>, 7));
result<127:0> = Vd<127:0> EOR tmp<127:0>;
tmp2 = ROL(tmp<31:0>, 15);
tmp2 = tmp2 EOR ROL(tmp2, 15) EOR ROL(tmp2, 23);
result<127:96> = result<127:96> EOR tmp2;
V[d, 128] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

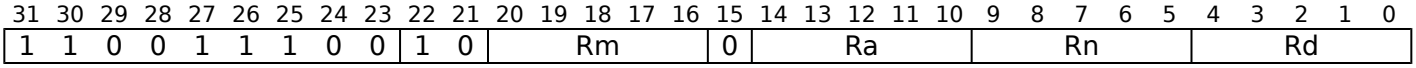
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SM3SS1

SM3SS1 rotates the top 32 bits of the 128-bit vector in the first source SIMD&FP register by 12, and adds that 32-bit value to the two other 32-bit values held in the top 32 bits of each of the 128-bit vectors in the second and third source SIMD&FP registers, rotating this result left by 7 and writing the final result into the top 32 bits of the vector in the destination SIMD&FP register, with the bottom 96 bits of the vector being written to 0.

This instruction is implemented only when *FEAT_SM3* is implemented.

Advanced SIMD
(FEAT_SM3)



SM3SS1 <Vd>.4S, <Vn>.4S, <Vm>.4S, <Va>.4S

```
if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer a = UInt(Ra);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn>	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
<Va>	Is the name of the third SIMD&FP source register, encoded in the "Ra" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
bits(128) Va = V[a, 128];
bits(128) result;
result<127:96> = ROL((ROL(Vn<127:96>, 12) + Vm<127:96> + Va<127:96>), 7);
result<95:0> = Zeros(96);
V[d, 128] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SM3TT1A

SM3TT1A takes three 128-bit vectors from three source SIMD&FP registers and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a three-way exclusive OR of the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

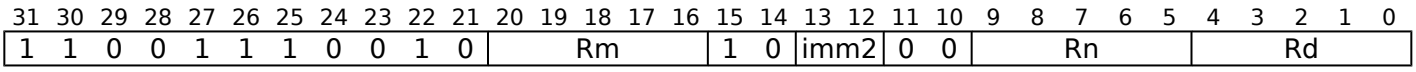
- The bottom 32-bit element of the first source vector, Vd, that was used for the three-way exclusive OR.
- The result of the exclusive OR of the top 32-bit element of the second source vector, Vn, with a rotation left by 12 of the top 32-bit element of the first source vector.
- A 32-bit element indexed out of the third source vector, Vm.

The result of this addition is returned as the top element of the result. The other elements of the result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 9.

This instruction is implemented only when *FEAT_SM3* is implemented.

Advanced SIMD

(FEAT_SM3)



SM3TT1A <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]

```
if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer i = UInt(imm2);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
<imm2>	Is a 32-bit element indexed out of <Vm>, encoded in "imm2".

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
bits(128) Vd = V[d, 128];
bits(32) WjPrime;
bits(128) result;
bits(32) TT1;
bits(32) SS2;

WjPrime = Elem[Vm, i, 32];
SS2 = Vn<127:96> EOR ROL(Vd<127:96>, 12);
TT1 = Vd<63:32> EOR (Vd<127:96> EOR Vd<95:64>);
TT1 = (TT1+Vd<31:0>+SS2+WjPrime)<31:0>;
result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>, 9);
result<95:64> = Vd<127:96>;
result<127:96> = TT1;
V[d, 128] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SM3TT1B

SM3TT1B takes three 128-bit vectors from three source SIMD&FP registers and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a 32-bit majority function between the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

- The bottom 32-bit element of the first source vector, Vd, that was used for the 32-bit majority function.
- The result of the exclusive OR of the top 32-bit element of the second source vector, Vn, with a rotation left by 12 of the top 32-bit element of the first source vector.
- A 32-bit element indexed out of the third source vector, Vm.

The result of this addition is returned as the top element of the result. The other elements of the result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 9.

This instruction is implemented only when *FEAT_SM3* is implemented.

Advanced SIMD
(FEAT_SM3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	0	Rm					1	0	imm2		0	1	Rn					Rd				

SM3TT1B <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]

```
if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer i = UInt(imm2);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
<imm2>	Is a 32-bit element indexed out of <Vm>, encoded in "imm2".

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
bits(128) Vd = V[d, 128];
bits(32) WjPrime;
bits(128) result;
bits(32) TT1;
bits(32) SS2;

WjPrime = Elem[Vm, i, 32];
SS2 = Vn<127:96> EOR ROL(Vd<127:96>, 12);
TT1 = (Vd<127:96> AND Vd<63:32>) OR (Vd<127:96> AND Vd<95:64>) OR (Vd<63:32> AND Vd<95:64>);
TT1 = (TT1+Vd<31:0>+SS2+WjPrime)<31:0>;
result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>, 9);
result<95:64> = Vd<127:96>;
result<127:96> = TT1;
V[d, 128] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SM3TT2A

SM3TT2A takes three 128-bit vectors from three source SIMD&FP register and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a three-way exclusive OR of the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

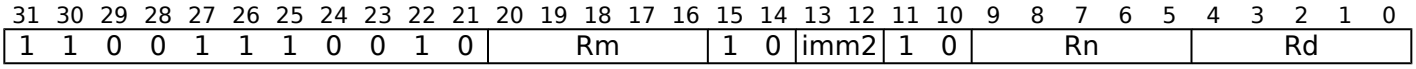
- The bottom 32-bit element of the first source vector, Vd, that was used for the three-way exclusive OR.
- The 32-bit element held in the top 32 bits of the second source vector, Vn.
- A 32-bit element indexed out of the third source vector, Vm.

A three-way exclusive OR is performed of the result of this addition, the result of the addition rotated left by 9, and the result of the addition rotated left by 17. The result of this exclusive OR is returned as the top element of the returned result. The other elements of this result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 19.

This instruction is implemented only when *FEAT_SM3* is implemented.

Advanced SIMD

(FEAT_SM3)



SM3TT2A <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]

```
if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer i = UInt(imm2);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
<imm2>	Is a 32-bit element indexed out of <Vm>, encoded in "imm2".

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
bits(128) Vd = V[d, 128];
bits(32) Wj;
bits(128) result;
bits(32) TT2;

Wj = Elem[Vm, i, 32];
TT2 = Vd<63:32> EOR (Vd<127:96> EOR Vd<95:64>);
TT2 = (TT2+Vd<31:0>+Vn<127:96>+Wj)<31:0>;

result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>, 19);
result<95:64> = Vd<127:96>;
result<127:96> = TT2 EOR ROL(TT2, 9) EOR ROL(TT2, 17);
V[d, 128] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SM3TT2B

SM3TT2B takes three 128-bit vectors from three source SIMD&FP registers, and a 2-bit immediate index value, and returns a 128-bit result in the destination SIMD&FP register. It performs a 32-bit majority function between the three 32-bit fields held in the upper three elements of the first source vector, and adds the resulting 32-bit value and the following three other 32-bit values:

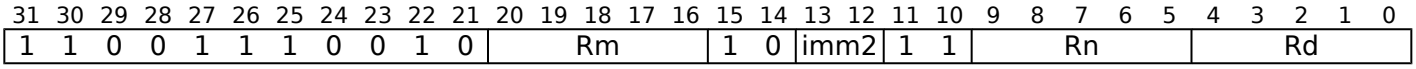
- The bottom 32-bit element of the first source vector, Vd, that was used for the 32-bit majority function.
- The 32-bit element held in the top 32 bits of the second source vector, Vn.
- A 32-bit element indexed out of the third source vector, Vm.

A three-way exclusive OR is performed of the result of this addition, the result of the addition rotated left by 9, and the result of the addition rotated left by 17. The result of this exclusive OR is returned as the top element of the returned result. The other elements of this result are taken from elements of the first source vector, with the element returned in bits<63:32> being rotated left by 19.

This instruction is implemented only when *FEAT_SM3* is implemented.

Advanced SIMD

(FEAT_SM3)



SM3TT2B <Vd>.4S, <Vn>.4S, <Vm>.S[<imm2>]

```
if !HaveSM3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer i = UInt(imm2);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the third SIMD&FP source register, encoded in the "Rm" field.
<imm2>	Is a 32-bit element indexed out of <Vm>, encoded in "imm2".

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
bits(128) Vd = V[d, 128];
bits(32) Wj;
bits(128) result;
bits(32) TT2;

Wj = Elem[Vm, i, 32];
TT2 = (Vd<127:96> AND Vd<95:64>) OR (NOT(Vd<127:96>) AND Vd<63:32>);
TT2 = (TT2+Vd<31:0>+Vn<127:96>+Wj)<31:0>;

result<31:0> = Vd<63:32>;
result<63:32> = ROL(Vd<95:64>, 19);
result<95:64> = Vd<127:96>;
result<127:96> = TT2 EOR ROL(TT2, 9) EOR ROL(TT2, 17);
V[d, 128] = result;
```

Operational information

If PSTATE.DIT is 1:

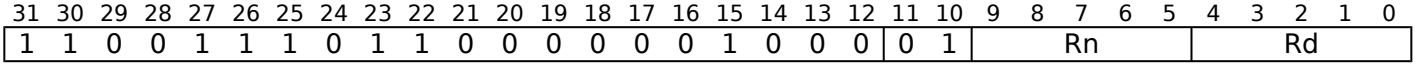
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SM4E

SM4 Encode takes input data as a 128-bit vector from the first source SIMD&FP register, and four iterations of the round key held as the elements of the 128-bit vector in the second source SIMD&FP register. It encrypts the data by four rounds, in accordance with the SM4 standard, returning the 128-bit result to the destination SIMD&FP register. This instruction is implemented only when *FEAT_SM4* is implemented.

Advanced SIMD
(FEAT_SM4)



SM4E <Vd>.4S, <Vn>.4S

```
if !HaveSM4Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the second SIMD&FP source register, encoded in the "Rn" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vn = V[n, 128];
bits(32) intval;
bits(128) roundresult;
bits(32) roundkey;

roundresult = V[d, 128];
for index = 0 to 3
    roundkey = Elem[Vn, index, 32];

    intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR roundkey;

    for i = 0 to 3
        Elem[intval, i, 8] = Sbox(Elem[intval, i, 8]);

    intval = intval EOR ROL(intval, 2) EOR ROL(intval, 10) EOR ROL(intval, 18) EOR ROL(intval, 24);
    intval = intval EOR roundresult<31:0>;

    roundresult<31:0> = roundresult<63:32>;
    roundresult<63:32> = roundresult<95:64>;
    roundresult<95:64> = roundresult<127:96>;
    roundresult<127:96> = intval;

V[d, 128] = roundresult;
```

Operational information

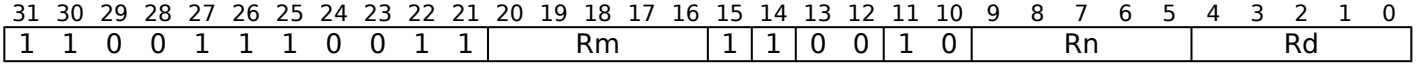
If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

SM4EKEY

SM4 Key takes an input as a 128-bit vector from the first source SIMD&FP register and a 128-bit constant from the second SIMD&FP register. It derives four iterations of the output key, in accordance with the SM4 standard, returning the 128-bit result to the destination SIMD&FP register.
This instruction is implemented only when *FEAT_SM4* is implemented.

Advanced SIMD
(FEAT_SM4)



SM4EKEY <Vd>.4S, <Vn>.4S, <Vm>.4S

```
if !HaveSM4Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn>	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(32) intval;
bits(128) result;
bits(32) const;
bits(128) roundresult;

roundresult = V[n, 128];
for index = 0 to 3
    const = Elem[Vm, index, 32];

    intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR const;

    for i = 0 to 3
        Elem[intval, i, 8] = Sbox(Elem[intval, i, 8]);

    intval = intval EOR ROL(intval, 13) EOR ROL(intval, 23);
    intval = intval EOR roundresult<31:0>;

    roundresult<31:0> = roundresult<63:32>;
    roundresult<63:32> = roundresult<95:64>;
    roundresult<95:64> = roundresult<127:96>;
    roundresult<127:96> = intval;

V[d, 128] = roundresult;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

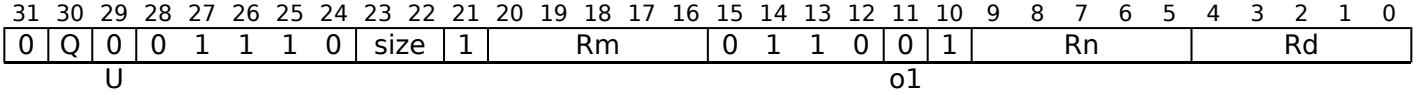
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMAX

Signed Maximum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the larger of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

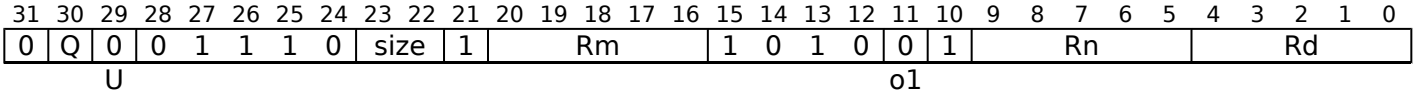
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMAXP

Signed Maximum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[concat, 2*e, esize], unsigned);
    element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

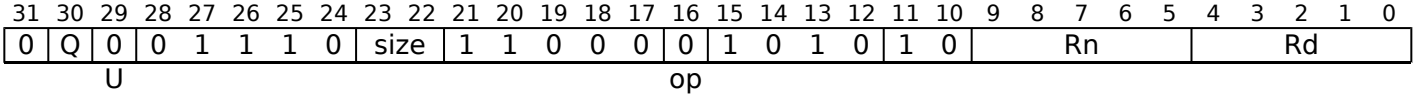
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMAXV

Signed Maximum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are signed integer values.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SMAXV <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');
```

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
integer maxmin;
integer element;

maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);

V[d, esize] = maxmin<esize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

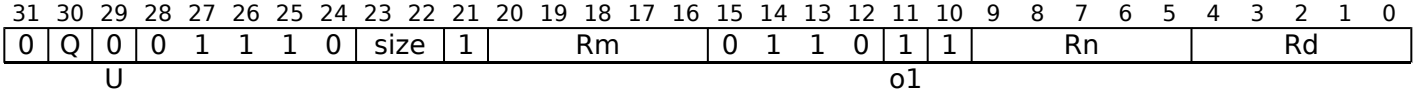
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMIN

Signed Minimum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the smaller of each of the two signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

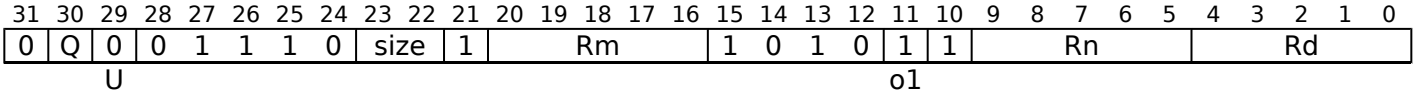
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMINP

Signed Minimum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of signed integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[concat, 2*e, esize], unsigned);
    element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

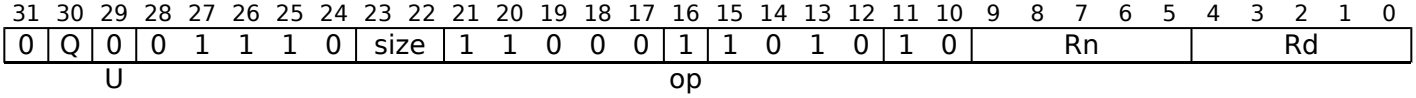
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMINV

Signed Minimum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are signed integer values.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SMINV <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');
```

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
integer maxmin;
integer element;

maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);

V[d, esize] = maxmin<esize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

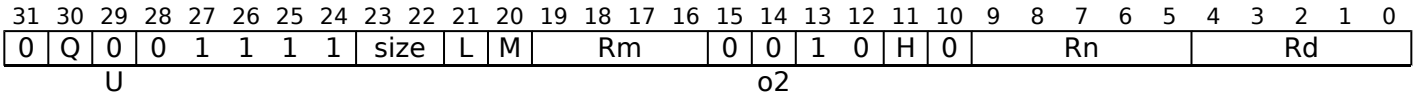
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMLAL, SMLAL2 (by element)

Signed Multiply-Add Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element in the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are signed integer values.

The SMLAL instruction extracts vector elements from the lower half of the first source register. The SMLAL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d, 2*datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

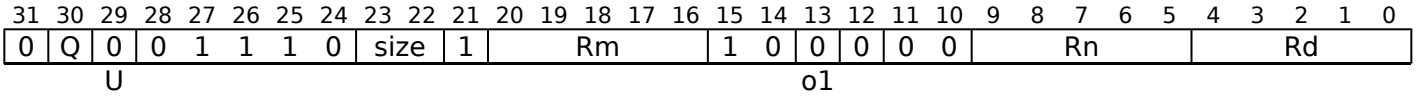
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

SMLAL, SMLAL2 (vector)

Signed Multiply-Add Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMLAL instruction extracts each source vector from the lower half of each source register. The SMLAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.




```
SMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d, 2*datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

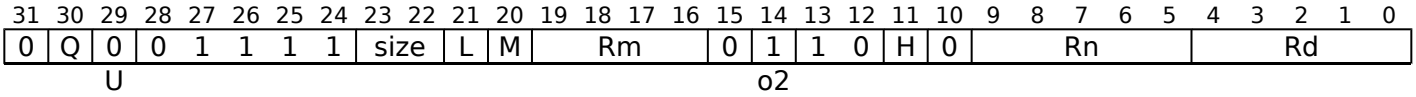
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMLSL, SMLSL2 (by element)

Signed Multiply-Subtract Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMLSL instruction extracts vector elements from the lower half of the first source register. The SMLSL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d, 2*datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

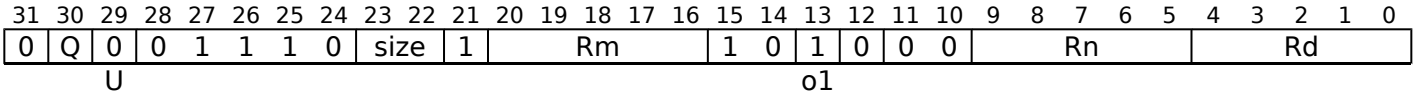
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

SMLSL, SMLSL2 (vector)

Signed Multiply-Subtract Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMLSL instruction extracts each source vector from the lower half of each source register. The SMLSL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d, 2*datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

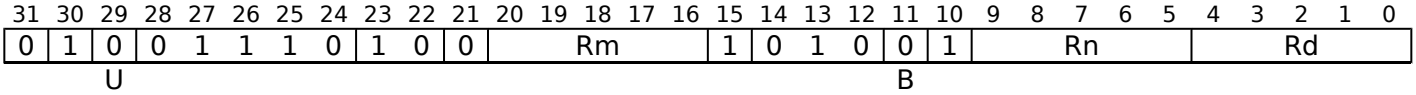
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMMLA (vector)

Signed 8-bit integer matrix multiply-accumulate. This instruction multiplies the 2x8 matrix of signed 8-bit integer values in the first source vector by the 8x2 matrix of signed 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2 to Armv8.5, this is an OPTIONAL instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. *ID_AA64ISAR1_EL1*.I8MM indicates whether this instruction is supported.

Vector
(FEAT_I8MM)



SMMLA <Vd>.4S, <Vn>.16B, <Vm>.16B

```
if !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

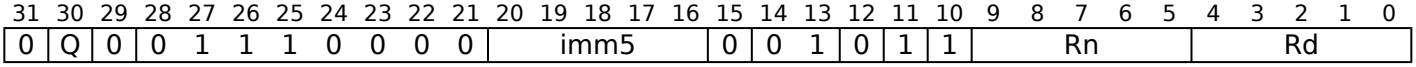
```
CheckFPAdvSIMDEnabled64();
bits(128) operand1 = V[n, 128];
bits(128) operand2 = V[m, 128];
bits(128) addend = V[d, 128];

V[d, 128] = MatMulAdd(addend, operand1, operand2, FALSE, FALSE);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMOV

Signed Move vector element to general-purpose register. This instruction reads the signed integer from the source SIMD&FP register, sign-extends it to form a 32-bit or 64-bit value, and writes the result to destination general-purpose register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



32-bit (Q == 0)

```
SMOV <Wd>, <Vn>.<Ts>[<index>]
```

64-bit (Q == 1)

```
SMOV <Xd>, <Vn>.<Ts>[<index>]
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size;
case Q:imm5 of
    when 'xxxxx1' size = 0;    // SMOV [WX]d, Vn.B
    when 'xxxx10' size = 1;    // SMOV [WX]d, Vn.H
    when '1xx100' size = 2;    // SMOV Xd, Vn.S
    otherwise UNDEFINED;

integer idxdsize = if imm5<4> == '1' then 128 else 64;
integer index = UInt(imm5<4:size+1>);
integer esize = 8 << size;
integer datasize = if Q == '1' then 64 else 32;
```

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ts> For the 32-bit variant: is an element size specifier, encoded in “imm5”:

imm5	<Ts>
xxx00	RESERVED
xxxx1	B
xxx10	H

For the 64-bit variant: is an element size specifier, encoded in “imm5”:

imm5	<Ts>
xx000	RESERVED
xxxx1	B
xxx10	H
xx100	S

<index> For the 32-bit variant: is the element index encoded in “imm5”:

imm5	<index>
xxx00	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>

For the 64-bit variant: is the element index encoded in “imm5”:

imm5	<index>
xx000	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>
xx100	imm5<4:3>

Operation

```
if index == 0 then
    CheckFPEnabled64();
else
    CheckFPAdvSIMDEnabled64();
bits(idxdsize) operand = V[n, idxdsize];

X[d, datasize] = SignExtend(Elem[operand, index, esize], datasize);
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

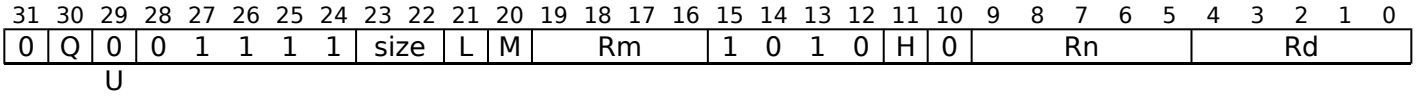
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMULL, SMULL2 (by element)

Signed Multiply Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, places the result in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The SMULL instruction extracts vector elements from the lower half of the first source register. The SMULL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = product;

V[d, 2*datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

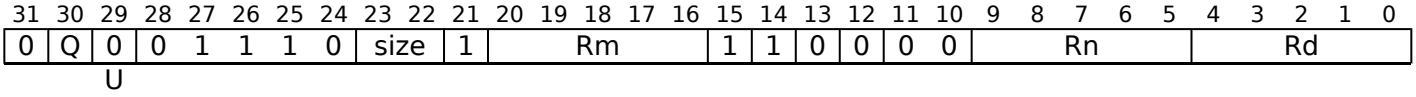
SMULL, SMULL2 (vector)

Signed Multiply Long (vector). This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, places the results in a vector, and writes the vector to the destination SIMD&FP register.

The destination vector elements are twice as long as the elements that are multiplied.

The SMULL instruction extracts each source vector from the lower half of each source register. The SMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, 2*esize] = (element1*element2)<2*esize-1:0>;

V[d, 2*datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQABS

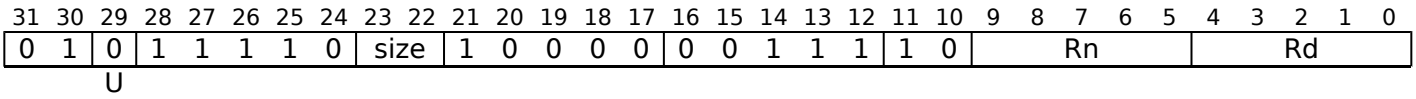
Signed saturating Absolute value. This instruction reads each vector element from the source SIMD&FP register, puts the absolute value of the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

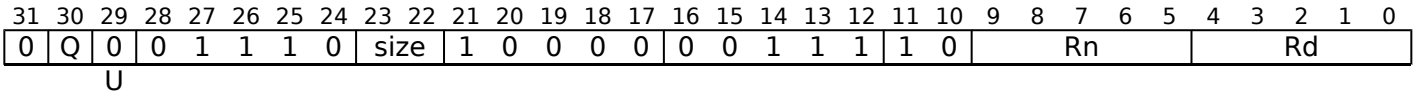
Scalar



SQABS <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');
```

Vector



SQABS <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    (Elem[result, e, esize], sat) = SignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQADD

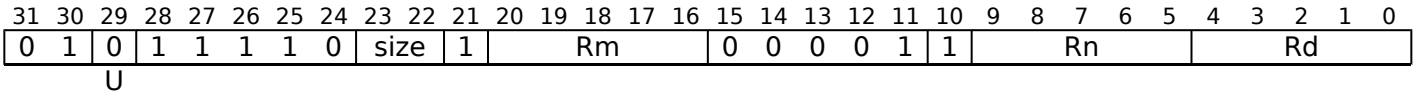
Signed saturating Add. This instruction adds the values of corresponding elements of the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

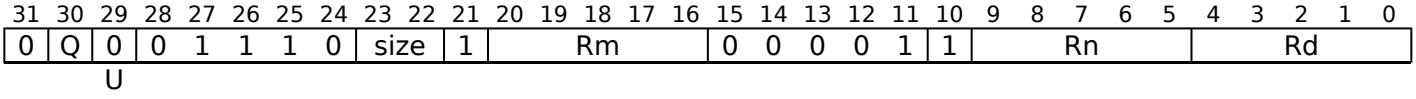
Scalar



SQADD <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Vector



SQADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer sum;
boolean sat;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = element1 + element2;
    (Elem[result, e, esize], sat) = SatQ(sum, esize, unsigned);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQDMLAL, SQDMLAL2 (by element)

Signed saturating Doubling Multiply-Add Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, and accumulates the final results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

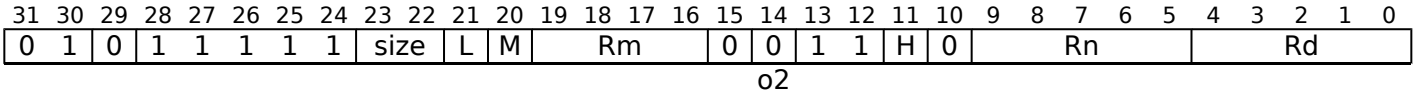
If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

The SQDMLAL instruction extracts vector elements from the lower half of the first source register. The SQDMLAL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

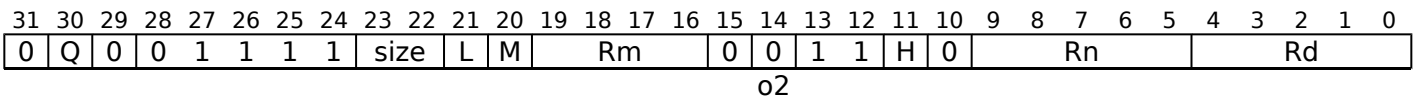
Scalar



SQDMLAL <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o2 == '1');
```

Vector



SQDMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Va> Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d, 2*datasize] = result;
```

SQDMLAL, SQDMLAL2 (vector)

Signed saturating Doubling Multiply-Add Long. This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, doubles the results, and accumulates the final results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

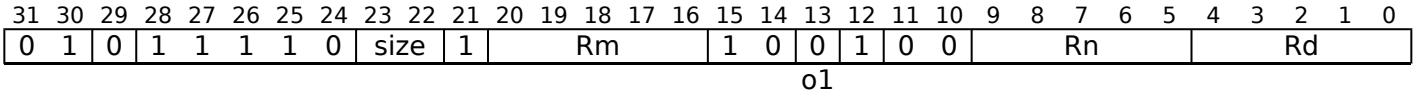
If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

The SQDMLAL instruction extracts each source vector from the lower half of each source register. The SQDMLAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

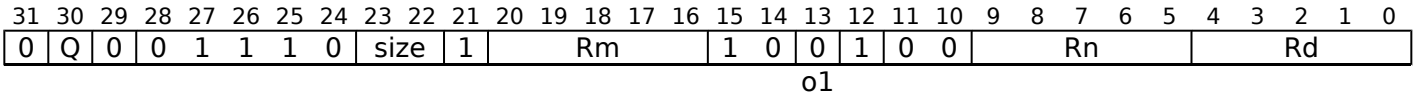
Scalar



SQDMLAL <Va><d>, <Vb><n>, <Vb><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o1 == '1');
```

Vector



SQDMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
```

Assembler Symbols

2	Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Va> Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d, 2*datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQDMLSL, SQDMLSL2 (by element)

Signed saturating Doubling Multiply-Subtract Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, and subtracts the final results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are signed integer values.

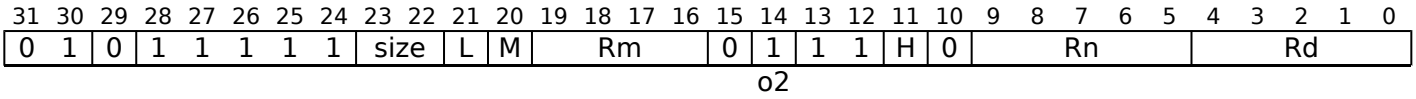
If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

The `SQDMLSL` instruction extracts vector elements from the lower half of the first source register. The `SQDMLSL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

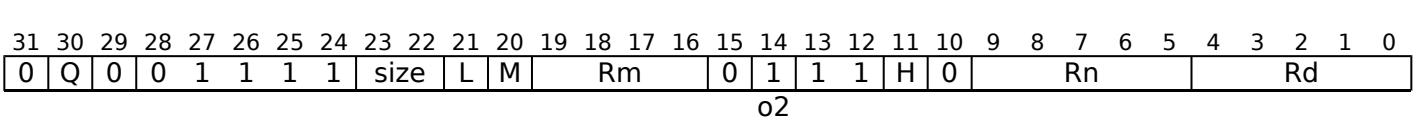
Scalar



SQDMLSL <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o2 == '1');
```

Vector



SQDMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o2 == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Va> Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d, 2*datasize] = result;
```

SQDMLSL, SQDMLSL2 (vector)

Signed saturating Doubling Multiply-Subtract Long. This instruction multiplies corresponding signed integer values in the lower or upper half of the vectors of the two source SIMD&FP registers, doubles the results, and subtracts the final results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

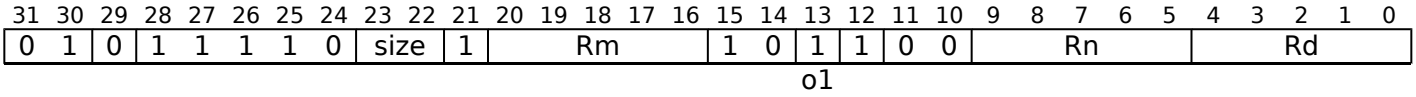
If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

The SQDMLSL instruction extracts each source vector from the lower half of each source register. The SQDMLSL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

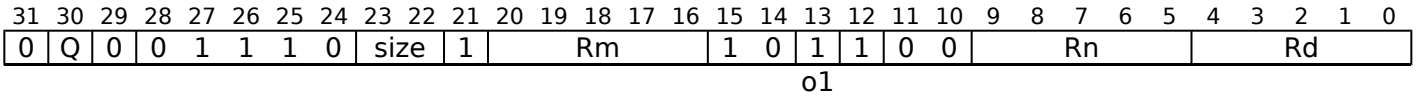
Scalar



SQDMLSL <Va><d>, <Vb><n>, <Vb><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;

boolean sub_op = (o1 == '1');
```

Vector



SQDMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
```

Assembler Symbols

2	Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Va> Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
integer accum;
boolean sat1;
boolean sat2;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat1) = SignedSatQ(2 * element1 * element2, 2 * esize);
    if sub_op then
        accum = SInt(Elem[operand3, e, 2*esize]) - SInt(product);
    else
        accum = SInt(Elem[operand3, e, 2*esize]) + SInt(product);
    (Elem[result, e, 2*esize], sat2) = SignedSatQ(accum, 2 * esize);
    if sat1 || sat2 then FPSR.QC = '1';

V[d, 2*datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQDMULH (by element)

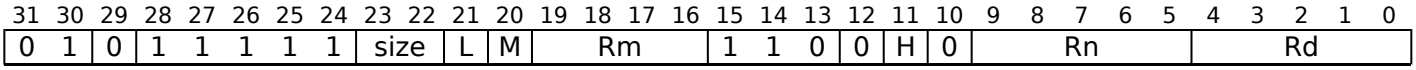
Signed saturating Doubling Multiply returning High half (by element). This instruction multiplies each vector element in the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see *SQRDMULH*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

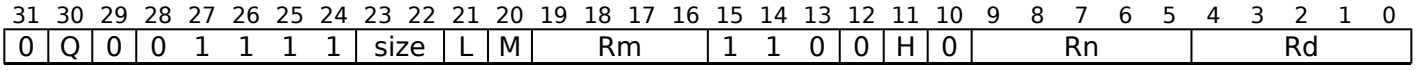
Scalar



SQDMULH <V><d>, <V><n>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean round = (op == '1');
```

Vector



SQDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean round = (op == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    product = (2 * element1 * element2) + round_const;
    // The following only saturates if element1 and element2 equal -(2^(esize-1))
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQDMULH (vector)

Signed saturating Doubling Multiply returning High half. This instruction multiplies the values of corresponding elements of the two source SIMD&FP registers, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

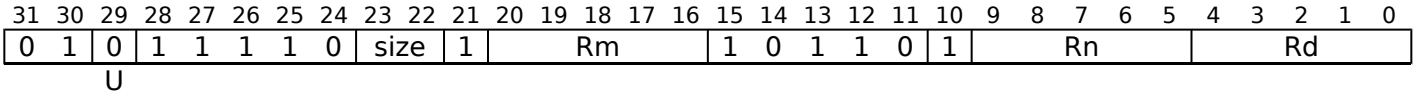
The results are truncated. For rounded results, see *SQRDMULH*.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

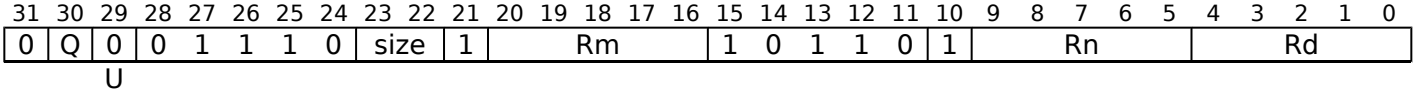
Scalar



SQDMULH <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = (U == '1');
```

Vector



SQDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer round_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    product = (2 * element1 * element2) + round_const;
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQDMULL, SQDMULL2 (by element)

Signed saturating Doubling Multiply Long (by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values.

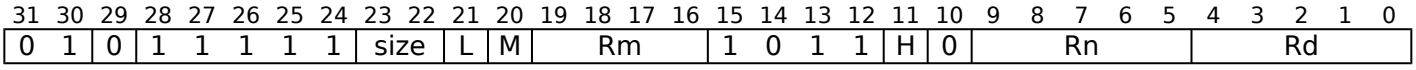
If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

The SQDMULL instruction extracts the first source vector from the lower half of the first source register. The SQDMULL2 instruction extracts the first source vector from the upper half of the first source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

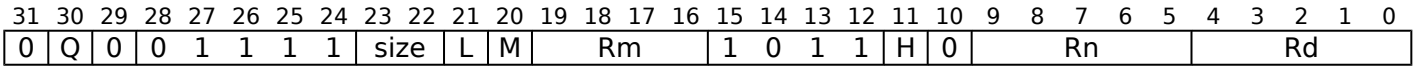
Scalar



SQDMULL <Va><d>, <Vb><n>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;
```

Vector



SQDMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Va> Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) operand1 = Vpart[n, part, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    (product, sat) = SignedSatQ(2 * element1 * element2, 2 * esize);
    Elem[result, e, 2*esize] = product;
    if sat then FPSR.QC = '1';

V[d, 2*datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQDMULL, SQDMULL2 (vector)

Signed saturating Doubling Multiply Long. This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, doubles the results, places the final results in a vector, and writes the vector to the destination SIMD&FP register.

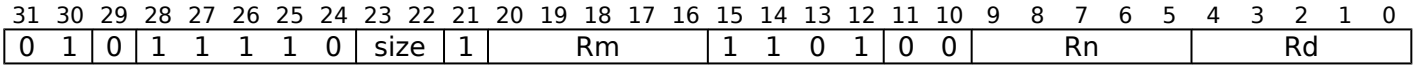
If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

The SQDMULL instruction extracts each source vector from the lower half of each source register. The SQDMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

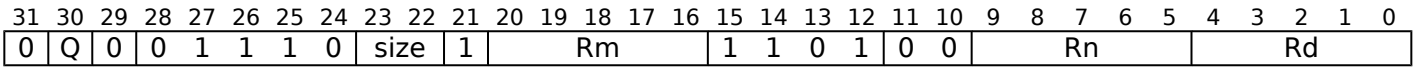
Scalar



SQDMULL <Va><d>, <Vb><n>, <Vb><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
integer part = 0;
```

Vector



SQDMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '00' || size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Va> Is the destination width specifier, encoded in “size”:

size	<Va>
00	RESERVED
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vb> Is the source width specifier, encoded in “size”:

size	<Vb>
00	RESERVED
01	H
10	S
11	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    (product, sat) = SignedSatQ(2 * element1 * element2, 2 * esize);
    Elem[result, e, 2*esize] = product;
    if sat then FPSR.QC = '1';

V[d, 2*datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQNEG

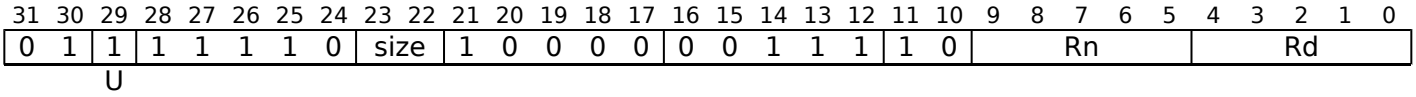
Signed saturating Negate. This instruction reads each vector element from the source SIMD&FP register, negates each value, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

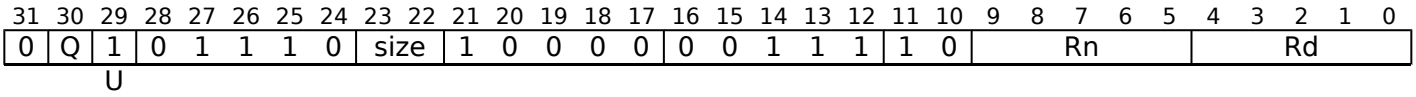
Scalar



SQNEG <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean neg = (U == '1');
```

Vector



SQNEG <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean neg = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = SInt(Elem[operand, e, esize]);
    if neg then
        element = -element;
    else
        element = Abs(element);
    (Elem[result, e, esize], sat) = SignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQRDMLAH (by element)

Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (by element). This instruction multiplies the vector elements of the first source SIMD&FP register with the value of a vector element of the second source SIMD&FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded.

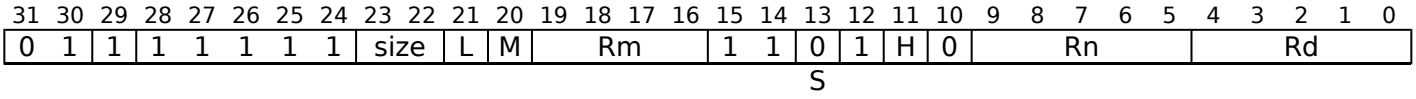
If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSR*.QC, is set if saturation occurs.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_RDM)

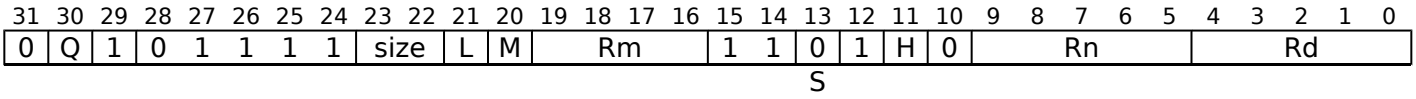


SQRDMLAH <V><d>, <V><n>, <Vm>.<Ts>[<index>]

```
if !HaveQRDMLAHExt() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Vector

(FEAT_RDM)



SQRDMLAH <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

```
if !HaveQRDMLAHExt() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
integer accum;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQRDMLAH (vector)

Signed Saturating Rounding Doubling Multiply Accumulate returning High Half (vector). This instruction multiplies the vector elements of the first source SIMD&FP register with the corresponding vector elements of the second source SIMD&FP register without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded.

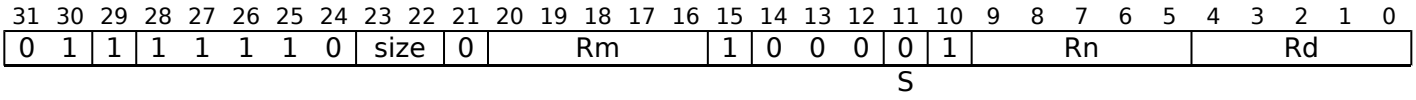
If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSR*.QC, is set if saturation occurs.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_RDM)

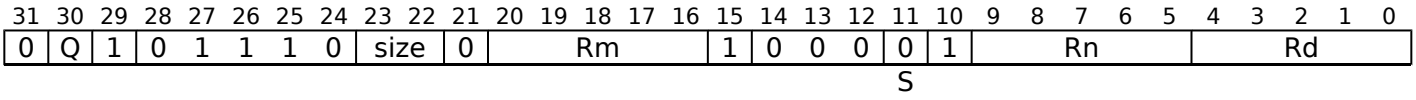


SQRDMLAH <V><d>, <V><n>, <V><m>

```
if !HaveQRDMLAHExt() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Vector

(FEAT_RDM)



SQRDMLAH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveQRDMLAHExt() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
integer accum;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

SQRDMLSH (by element)

Signed Saturating Rounding Doubling Multiply Subtract returning High Half (by element). This instruction multiplies the vector elements of the first source SIMD&FP register with the value of a vector element of the second source SIMD&FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD&FP register. The results are rounded.

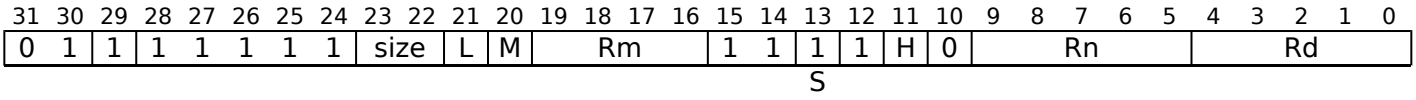
If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSR*.QC, is set if saturation occurs.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_RDM)

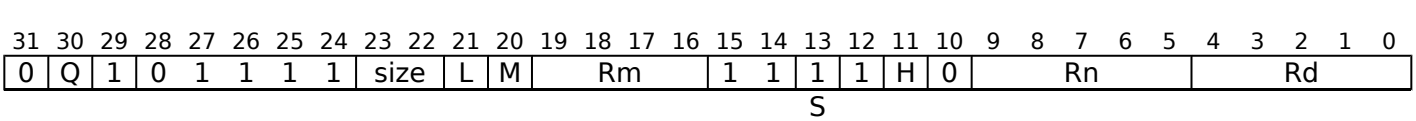


SQRDMLSH <V><d>, <V><n>, <Vm>.<Ts>[<index>]

```
if !HaveQRDMLAHExt() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Vector

(FEAT_RDM)



SQRDMLSH <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

```
if !HaveQRDMLAHExt() then UNDEFINED;

integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
integer accum;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQRDMLSH (vector)

Signed Saturating Rounding Doubling Multiply Subtract returning High Half (vector). This instruction multiplies the vector elements of the first source SIMD&FP register with the corresponding vector elements of the second source SIMD&FP register without saturating the multiply results, doubles the results, and subtracts the most significant half of the final results from the vector elements of the destination SIMD&FP register. The results are rounded.

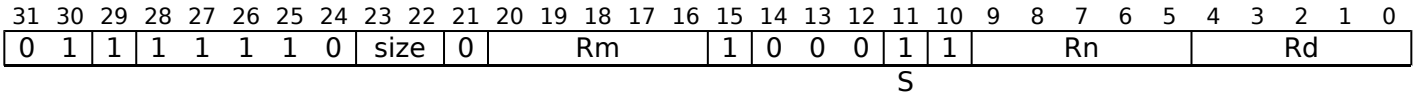
If any of the results overflow, they are saturated. The cumulative saturation bit, *FPSR*.QC, is set if saturation occurs.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

(FEAT_RDM)

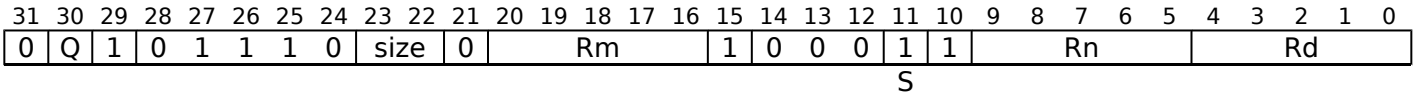


SQRDMLSH <V><d>, <V><n>, <V><m>

```
if !HaveQRDMLAHExt() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Vector

(FEAT_RDM)



SQRDMLSH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
if !HaveQRDMLAHExt() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = TRUE;
boolean sub_op = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
integer rounding_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer element3;
integer product;
integer accum;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    element3 = SInt(Elem[operand3, e, esize]);
    if sub_op then
        accum = ((element3 << esize) - 2 * (element1 * element2) + rounding_const);
    else
        accum = ((element3 << esize) + 2 * (element1 * element2) + rounding_const);
    (Elem[result, e, esize], sat) = SignedSatQ(accum >> esize, esize);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```

SQRDMULH (by element)

Signed saturating Rounding Doubling Multiply returning High half (by element). This instruction multiplies each vector element in the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

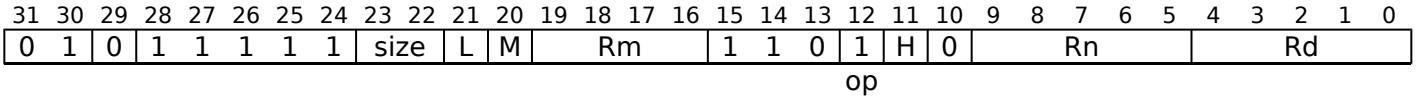
The results are rounded. For truncated results, see *SQDMULH*.

If any of the results overflows, they are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

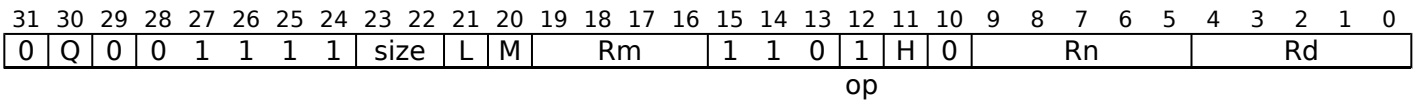
Scalar



SQRDMULH <V><d>, <V><n>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean round = (op == '1');
```

Vector



SQRDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean round = (op == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;

element2 = SInt(Elem[operand2, index, esize]);
for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    product = (2 * element1 * element2) + round_const;
    // The following only saturates if element1 and element2 equal -(2^(esize-1))
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQRDMULH (vector)

Signed saturating Rounding Doubling Multiply returning High half. This instruction multiplies the values of corresponding elements of the two source SIMD&FP registers, doubles the results, places the most significant half of the final results into a vector, and writes the vector to the destination SIMD&FP register.

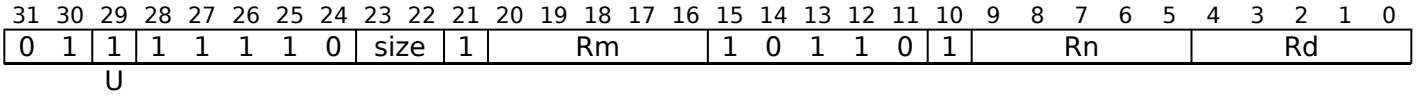
The results are rounded. For truncated results, see *SQDMULH*.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

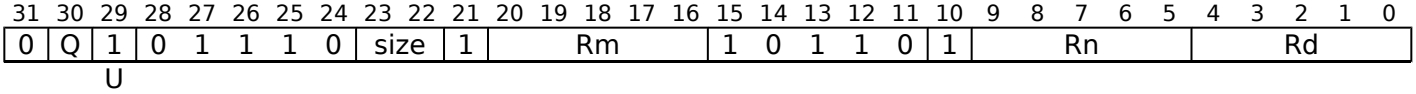
Scalar



SQRDMULH <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean rounding = (U == '1');
```

Vector



SQRDMULH <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' || size == '00' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean rounding = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer round_const = if rounding then 1 << (esize - 1) else 0;
integer element1;
integer element2;
integer product;
boolean sat;

for e = 0 to elements-1
    element1 = SInt(Elem[operand1, e, esize]);
    element2 = SInt(Elem[operand2, e, esize]);
    product = (2 * element1 * element2) + round_const;
    (Elem[result, e, esize], sat) = SignedSatQ(product >> esize, esize);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQRSHL

Signed saturating Rounding Shift Left (register). This instruction takes each vector element in the first source SIMD&FP register, shifts it by a value from the least significant byte of the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

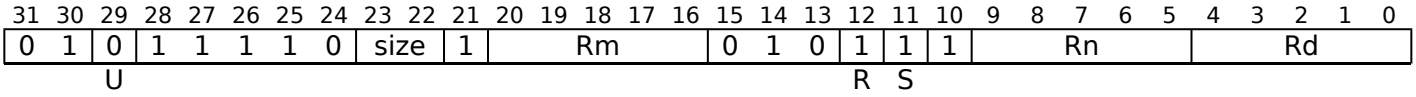
If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. For truncated results, see *SQSHL*.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

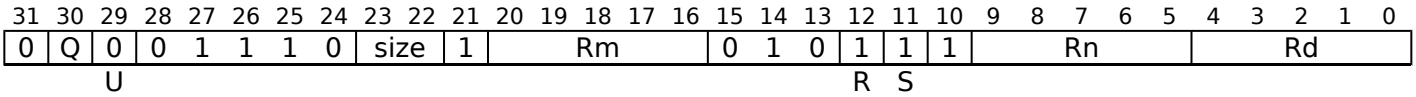
Scalar



SQRSHL <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;
```

Vector



SQRSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

boolean sat;

for e = 0 to elements-1
    integer element = Int(Elem[operand1, e, esize], unsigned);
    integer shift = SInt(Elem[operand2, e, esize]<7:0>);
    if shift >= 0 then // left shift
        element = element << shift;
    else // right shift
        shift = -shift;
        if rounding then
            element = (element + (1 << (shift - 1))) >> shift;
        else
            element = element >> shift;

    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;
```

SQRSHRN, SQRSHRN2

Signed saturating Rounded Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are half as long as the source vector elements. The results are rounded. For truncated results, see *SQSHRN*.

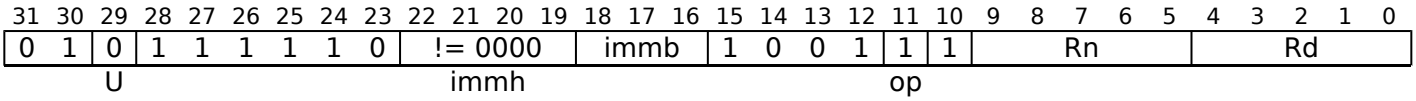
The SQRSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQRSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

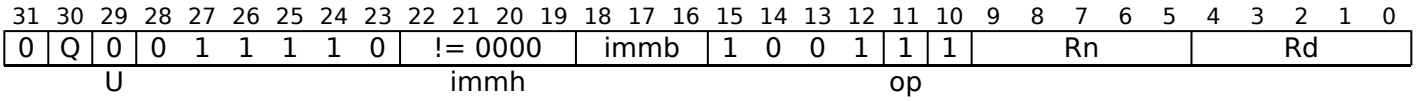
Scalar



SQRSHRN <Vb><d>, <Va><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

Vector



SQRSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vb> Is the destination width specifier, encoded in “immh”:

immh	<Vb>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in “immh”:

immh	<Va>
0000	RESERVED
0001	H
001x	S
01xx	D
1xxx	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

immh	<shift>
0000	RESERVED
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n, datasize*2];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
    element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQRSHRUN, SQRSHRUN2

Signed saturating Rounded Shift Right Unsigned Narrow (immediate). This instruction reads each signed integer value in the vector of the source SIMD&FP register, right shifts each value by an immediate value, saturates the result to an unsigned integer value that is half the original width, places the final result into a vector, and writes the vector to the destination SIMD&FP register. The results are rounded. For truncated results, see *SQSHRUN*.

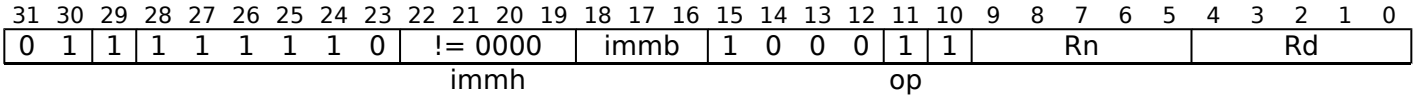
The SQRSHRUN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQRSHRUN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

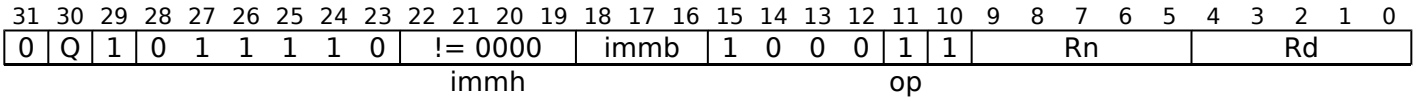
Scalar



SQRSHRUN <Vb><d>, <Va><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

Vector



SQRSHRUN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

Assembler Symbols

2	Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vb> Is the destination width specifier, encoded in “immh”:

immh	<Vb>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in “immh”:

immh	<Va>
0000	RESERVED
0001	H
001x	S
01xx	D
1xxx	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

immh	<shift>
0000	RESERVED
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n, datasize*2];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
    element = (SInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
    (Elem[result, e, esize], sat) = UnsignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

Vpart[d, part, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQSHL (immediate)

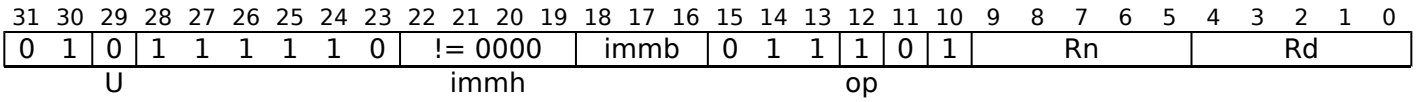
Signed saturating Shift Left (immediate). This instruction reads each vector element in the source SIMD&FP register, shifts each result by an immediate value, places the final result in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see *UQRSHL*.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

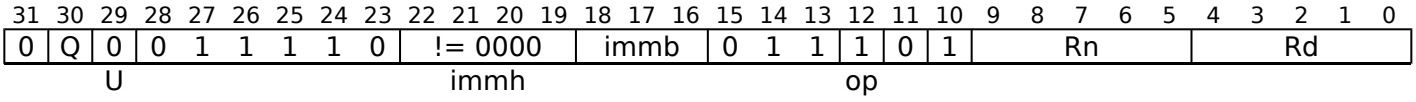
Scalar



SQSHL <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
    when '00' UNDEFINED;
    when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
    when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
    when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```

Vector



SQSHL <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
    when '00' UNDEFINED;
    when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
    when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
    when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	RESERVED
0001	(UInt(immh:immb)-8)
001x	(UInt(immh:immb)-16)
01xx	(UInt(immh:immb)-32)
1xxx	(UInt(immh:immb)-64)

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(UInt(immh:immb)-8)
001x	(UInt(immh:immb)-16)
01xx	(UInt(immh:immb)-32)
1xxx	(UInt(immh:immb)-64)

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], src_unsigned) << shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, dst_unsigned);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```

SQSHL (register)

Signed saturating Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts each element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

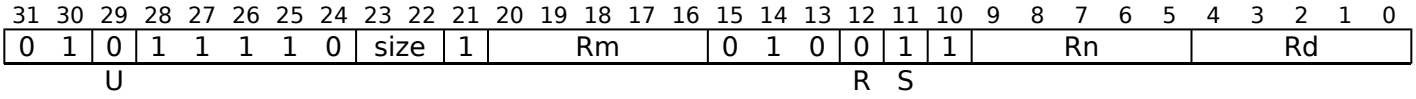
If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. For rounded results, see *SQRSHL*.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

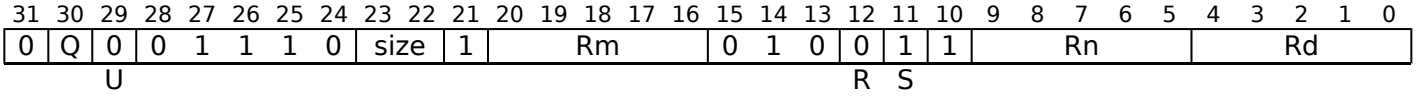
Scalar



SQSHL <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;
```

Vector



SQSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

boolean sat;

for e = 0 to elements-1
    integer element = Int(Elem[operand1, e, esize], unsigned);
    integer shift = SInt(Elem[operand2, e, esize]<7:0>);
    if shift >= 0 then // left shift
        element = element << shift;
    else // right shift
        shift = -shift;
        if rounding then
            element = (element + (1 << (shift - 1))) >> shift;
        else
            element = element >> shift;

    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQSHLU

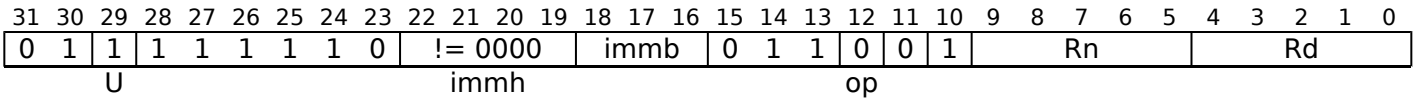
Signed saturating Shift Left Unsigned (immediate). This instruction reads each signed integer value in the vector of the source SIMD&FP register, shifts each value by an immediate value, saturates the shifted result to an unsigned integer value, places the result in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see *UQRSHL*.

If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

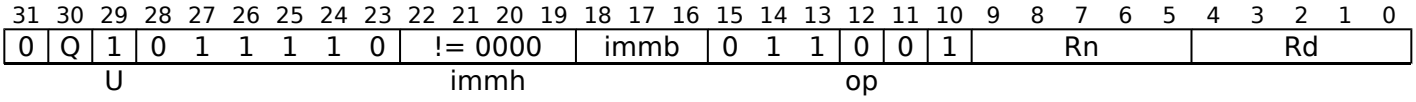
Scalar



SQSHLU <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
    when '00' UNDEFINED;
    when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
    when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
    when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```

Vector



SQSHLU <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
    when '00' UNDEFINED;
    when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
    when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
    when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	RESERVED
0001	(UInt(immh:immb)-8)
001x	(UInt(immh:immb)-16)
01xx	(UInt(immh:immb)-32)
1xxx	(UInt(immh:immb)-64)

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(UInt(immh:immb)-8)
001x	(UInt(immh:immb)-16)
01xx	(UInt(immh:immb)-32)
1xxx	(UInt(immh:immb)-64)

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], src_unsigned) << shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, dst_unsigned);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```

SQSHRN, SQSHRN2

Signed saturating Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts and truncates each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are half as long as the source vector elements. For rounded results, see *SQRSHRN*.

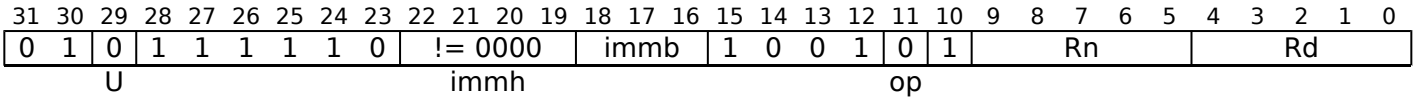
The SQSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

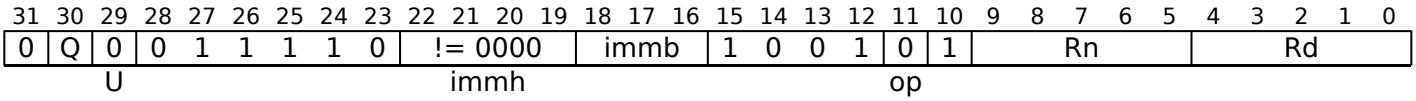
Scalar



SQSHRN <Vb><d>, <Va><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

Vector



SQSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vb> Is the destination width specifier, encoded in “immh”:

immh	<Vb>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in “immh”:

immh	<Va>
0000	RESERVED
0001	H
001x	S
01xx	D
1xxx	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

immh	<shift>
0000	RESERVED
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n, datasize*2];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
    element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQSHRUN, SQSHRUN2

Signed saturating Shift Right Unsigned Narrow (immediate). This instruction reads each signed integer value in the vector of the source SIMD&FP register, right shifts each value by an immediate value, saturates the result to an unsigned integer value that is half the original width, places the final result into a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see *SQRSHRUN*.

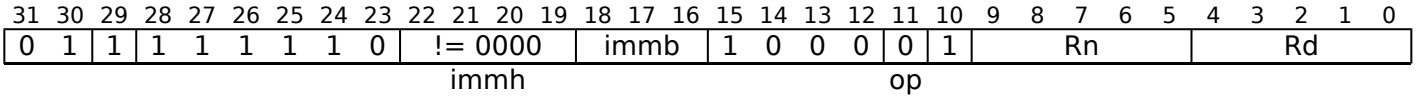
The SQSHRUN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQSHRUN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

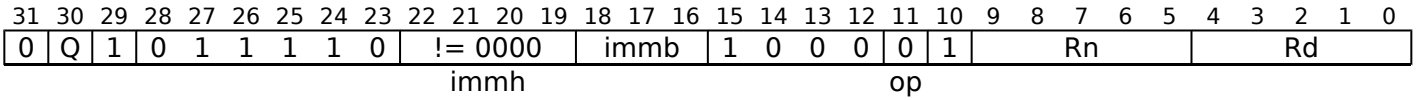
Scalar



SQSHRUN <Vb><d>, <Va><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

Vector



SQSHRUN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vb> Is the destination width specifier, encoded in “immh”:

immh	<Vb>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in “immh”:

immh	<Va>
0000	RESERVED
0001	H
001x	S
01xx	D
1xxx	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

immh	<shift>
0000	RESERVED
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n, datasize*2];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
    element = (SInt(Elem[operand, e, 2*esize]) + round_const) >> shift;
    (Elem[result, e, esize], sat) = UnsignedSatQ(element, esize);
    if sat then FPSR.QC = '1';

Vpart[d, part, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQSUB

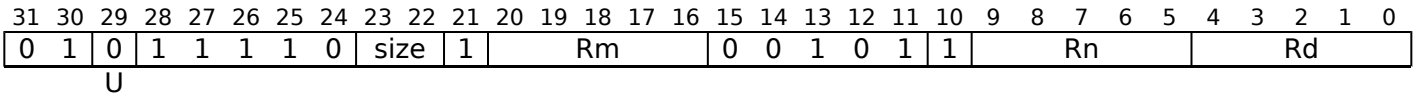
Signed saturating Subtract. This instruction subtracts the element values of the second source SIMD&FP register from the corresponding element values of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

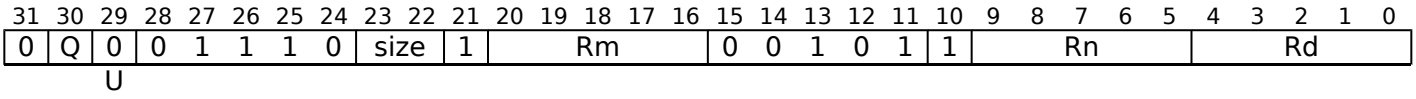
Scalar



SQSUB <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Vector



SQSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer diff;
boolean sat;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = element1 - element2;
    (Elem[result, e, esize], sat) = SatQ(diff, esize, unsigned);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQXTN, SQXTN2

Signed saturating extract Narrow. This instruction reads each vector element from the source SIMD&FP register, saturates the value to half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements. All the values in this instruction are signed integer values.

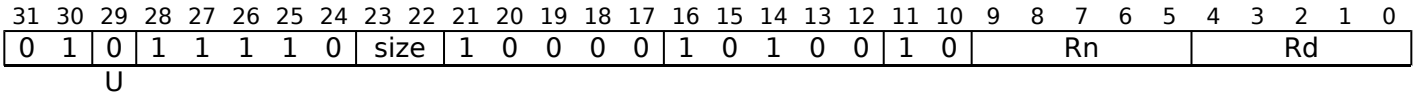
If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

The SQXTN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQXTN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

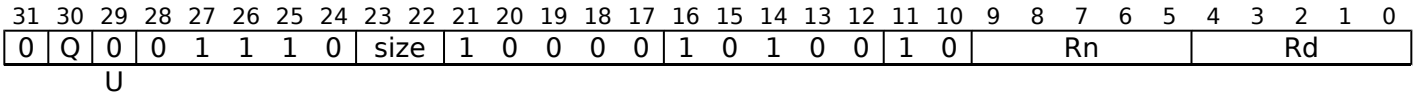
Scalar



SQXTN <Vb><d>, <Va><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer part = 0;
integer elements = 1;

boolean unsigned = (U == '1');
```

Vector



SQXTN{2} <Vd>.<Tb>, <Vn>.<Ta>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

2	Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vb> Is the destination width specifier, encoded in “size”:

size	<Vb>
00	B
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va> Is the source width specifier, encoded in “size”:

size	<Va>
00	H
01	S
10	D
11	RESERVED

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n, 2*datasize];
bits(datasize) result;
bits(2*esize) element;
boolean sat;

for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Elem[result, e, esize], sat) = SatQ(Int(element, unsigned), esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part, datasize] = result;
```

SQXTUN, SQXTUN2

Signed saturating extract Unsigned Narrow. This instruction reads each signed integer value in the vector of the source SIMD&FP register, saturates the value to an unsigned integer value that is half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements.

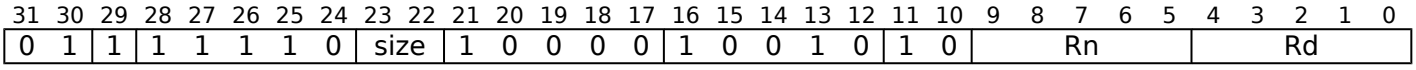
If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

The SQXTUN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SQXTUN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

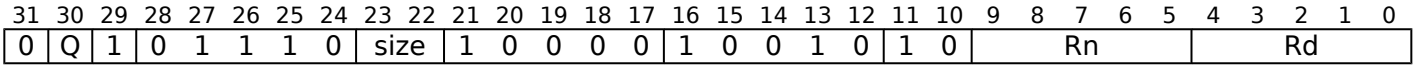
Scalar



SQXTUN <Vb><d>, <Va><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer part = 0;
integer elements = 1;
```

Vector



SQXTUN{2} <Vd>.<Tb>, <Vn>.<Ta>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vb> Is the destination width specifier, encoded in “size”:

size	<Vb>
00	B
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va> Is the source width specifier, encoded in “size”:

size	<Va>
00	H
01	S
10	D
11	RESERVED

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n, 2*datasize];
bits(datasize) result;
bits(2*esize) element;
boolean sat;

for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Elem[result, e, esize], sat) = UnsignedSatQ(SInt(element), esize);
    if sat then FPSR.QC = '1';

Vpart[d, part, datasize] = result;
```

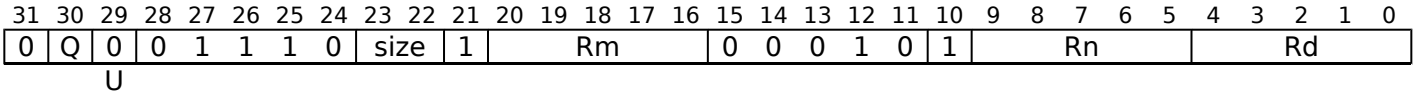
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SRHADD

Signed Rounding Halving Add. This instruction adds corresponding signed integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see *SHADD*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SRHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

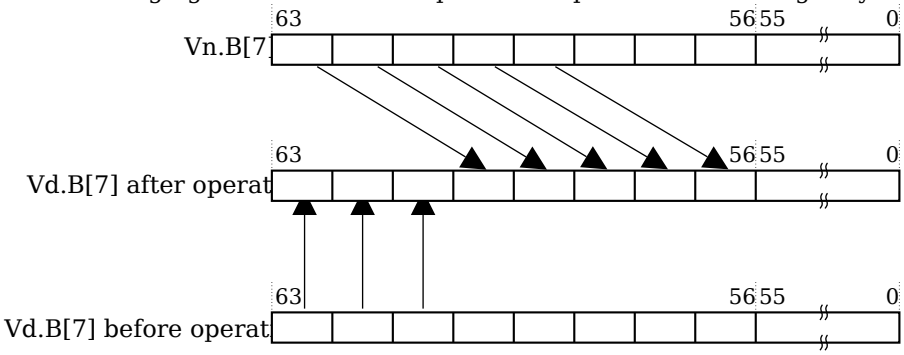
Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = (element1 + element2 + 1) >> 1;
    Elem[result, e, esize] = sum<esize-1:0>;

V[d, datasize] = result;
```

SRI

Shift Right and Insert (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each vector element by an immediate value, and inserts the result into the corresponding vector element in the destination SIMD&FP register such that the new zero bits created by the shift are not inserted but retain their existing value. Bits shifted out of the right of each vector element of the source register are lost.

The following figure shows an example of the operation of shift right by 3 for an 8-bit vector element.



Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0	!= 0000				immb			0	1	0	0	0	1	Rn					Rd				
									immh																						

SRI <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	1	0	!= 0000				immb			0	1	0	0	0	1	Rn					Rd				
									immh																						

SRI <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) operand2 = V[d, datasize];
bits(datasize) result;
bits(esize) mask = LSR(Ones(esize), shift);
bits(esize) shifted;

for e = 0 to elements-1
    shifted = LSR(Elem[operand, e, esize], shift);
    Elem[result, e, esize] = (Elem[operand2, e, esize] AND NOT(mask)) OR shifted;
V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

SRSHL

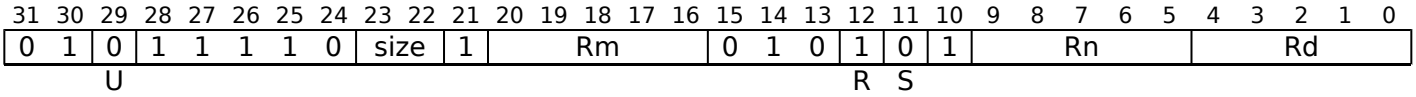
Signed Rounding Shift Left (register). This instruction takes each signed integer value in the vector of the first source SIMD&FP register, shifts it by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift. For a truncating shift, see *SSHL*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

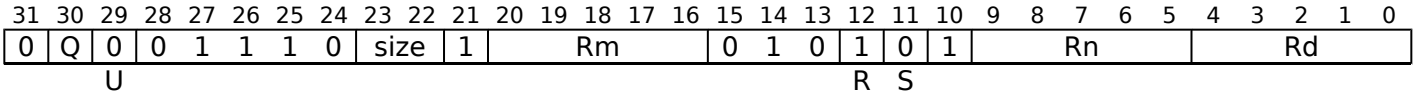
Scalar



SRSHL <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;
```

Vector



SRSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

boolean sat;

for e = 0 to elements-1
    integer element = Int(Elem[operand1, e, esize], unsigned);
    integer shift = SInt(Elem[operand2, e, esize]<7:0>);
    if shift >= 0 then // left shift
        element = element << shift;
    else // right shift
        shift = -shift;
        if rounding then
            element = (element + (1 << (shift - 1))) >> shift;
        else
            element = element >> shift;

    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

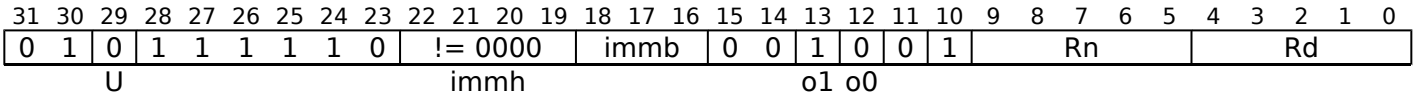
SRSHR

Signed Rounding Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, places the final result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are rounded. For truncated results, see *SSHR*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

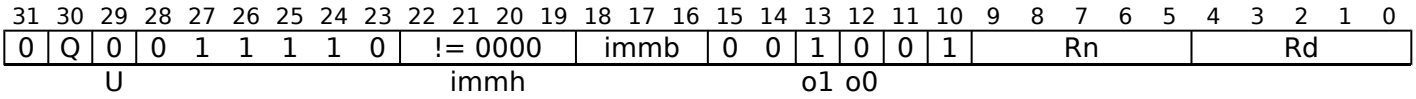
Scalar



SRSHR <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector



SRSHR <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

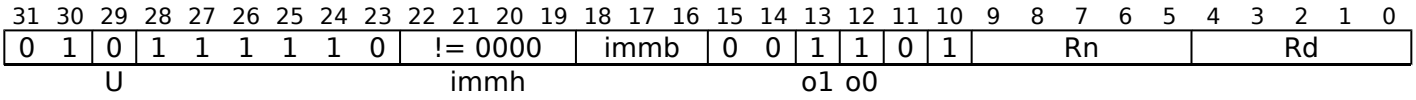
SRSRA

Signed Rounding Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are rounded. For truncated results, see *SSRA*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

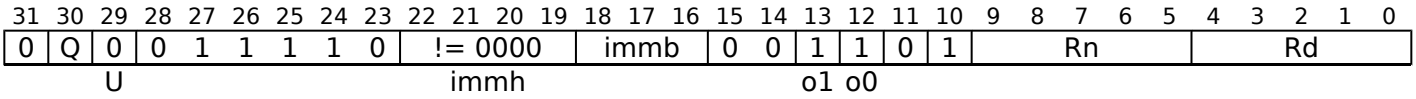
Scalar



SRSRA <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector



SRSRA <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SSHL

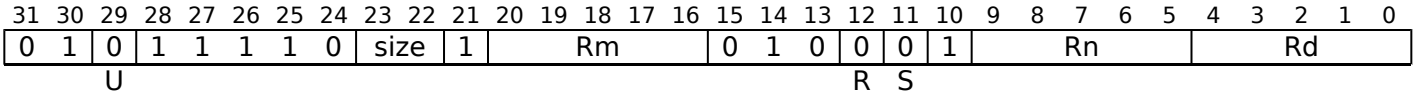
Signed Shift Left (register). This instruction takes each signed integer value in the vector of the first source SIMD&FP register, shifts each value by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. For a rounding shift, see *SRSHL*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

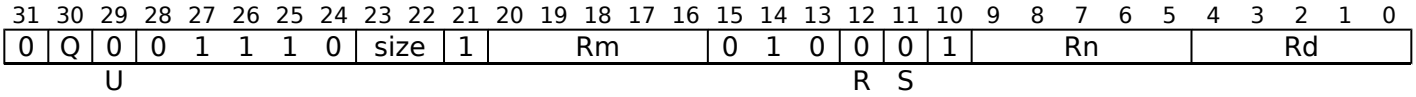
Scalar



SSHL <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;
```

Vector



SSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

boolean sat;

for e = 0 to elements-1
    integer element = Int(Elem[operand1, e, esize], unsigned);
    integer shift = SInt(Elem[operand2, e, esize]<7:0>);
    if shift >= 0 then // left shift
        element = element << shift;
    else // right shift
        shift = -shift;
        if rounding then
            element = (element + (1 << (shift - 1))) >> shift;
        else
            element = element >> shift;

    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

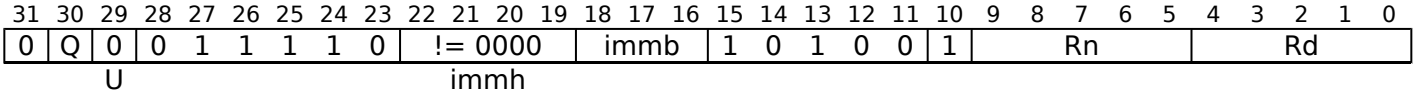
SSHLL, SSHLL2

Signed Shift Left Long (immediate). This instruction reads each vector element from the source SIMD&FP register, left shifts each vector element by the specified shift amount, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The SSHLL instruction extracts vector elements from the lower half of the source register. The SSHLL2 instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias SXTL, SXTL2.



SSHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<shift> Is the left shift amount, in the range 0 to the source element width in bits minus 1, encoded in "immh:immb":

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(UInt(immh:immb)-8)
001x	(UInt(immh:immb)-16)
01xx	(UInt(immh:immb)-32)
1xxx	RESERVED

Alias Conditions

Alias	Is preferred when
SXTL, SXTL2	immb == '000' && BitCount(immh) == 1

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part, datasize];
bits(datasize*2) result;
integer element;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], unsigned) << shift;
    Elem[result, e, 2*esize] = element<2*esize-1:0>;

V[d, datasize*2] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

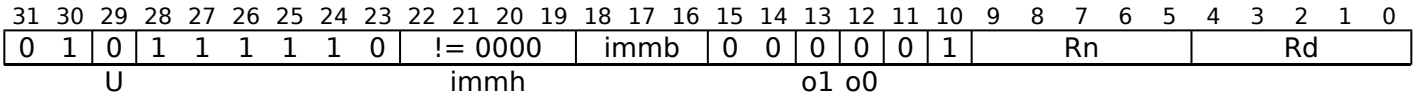
SSHR

Signed Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, places the final result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see *SRSHR*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

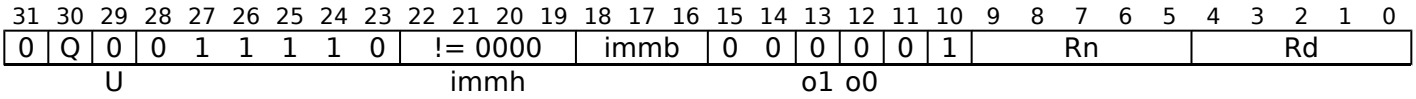
Scalar



SSHR <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector



SSHR <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

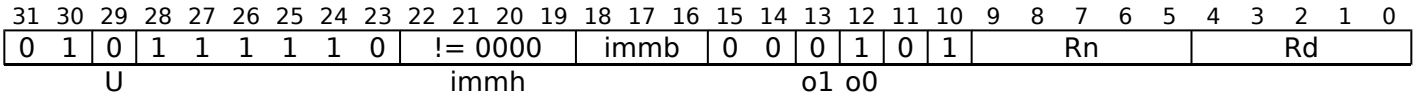
SSRA

Signed Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are signed integer values. The results are truncated. For rounded results, see *SRSRA*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

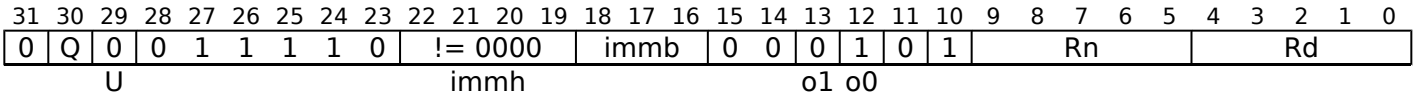
Scalar



SSRA <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector



SSRA <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

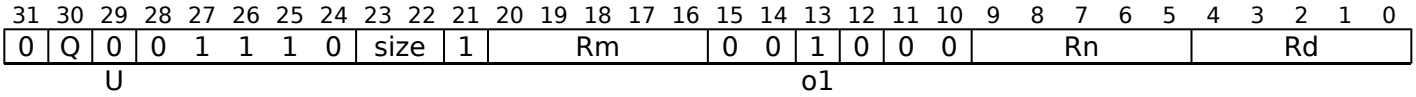
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SSUBL, SSUBL2

Signed Subtract Long. This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are signed integer values. The destination vector elements are twice as long as the source vector elements.

The SSUBL instruction extracts each source vector from the lower half of each source register. The SSUBL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SSUBL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d, 2*datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

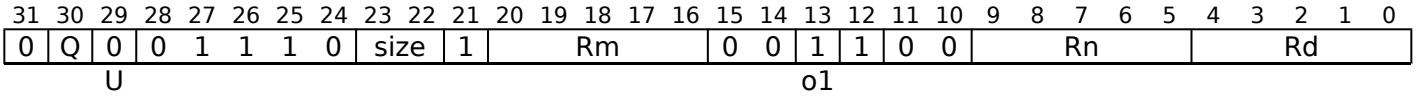
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SSUBW, SSUBW2

Signed Subtract Wide. This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. All the values in this instruction are signed integer values.

The SSUBW instruction extracts the second source vector from the lower half of the second source register. The SSUBW2 instruction extracts the second source vector from the upper half of the second source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SSUBW{2} <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n, 2*datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, 2*esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d, 2*datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

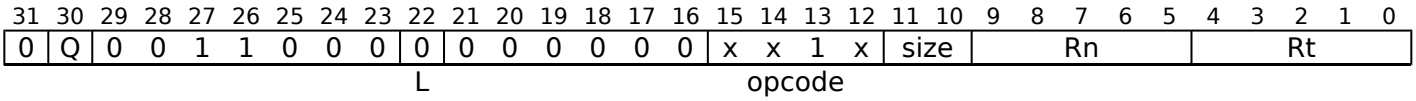
ST1 (multiple structures)

Store multiple single-element structures from one, two, three, or four registers. This instruction stores elements to memory from one, two, three, or four SIMD&FP registers, without interleaving. Every element of each register is stored.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset



One register (opcode == 0111)

```
ST1 { <Vt>.<T> }, [<Xn|SP>]
```

Two registers (opcode == 1010)

```
ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]
```

Three registers (opcode == 0110)

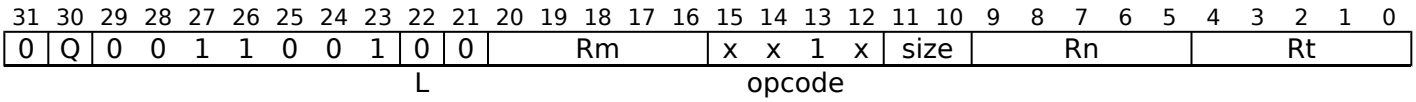
```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]
```

Four registers (opcode == 0010)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index



One register, immediate offset (Rm == 11111 && opcode == 0111)

```
ST1 { <Vt>.<T> }, [<Xn|SP>], <imm>
```

One register, register offset (Rm != 11111 && opcode == 0111)

```
ST1 { <Vt>.<T> }, [<Xn|SP>], <Xm>
```

Two registers, immediate offset (Rm == 11111 && opcode == 1010)

```
ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Two registers, register offset (Rm != 11111 && opcode == 1010)

```
ST1 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

Three registers, immediate offset (Rm == 11111 && opcode == 0110)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>
```

Three registers, register offset (Rm != 11111 && opcode == 0110)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>
```

Four registers, immediate offset (Rm == 11111 && opcode == 0010)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>
```

Four registers, register offset (Rm != 11111 && opcode == 0010)

```
ST1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> For the one register, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#8
1	#16

For the two registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#16
1	#32

For the three registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#24
1	#48

For the four registers, immediate offset variant: is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#32
1	#64

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;    // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
    otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt, datasize];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt, datasize] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

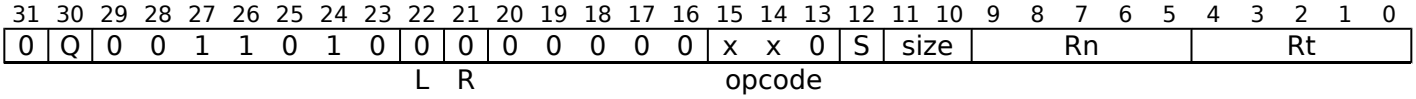
ST1 (single structure)

Store a single-element structure from one lane of one register. This instruction stores the specified element of a SIMD&FP register to memory.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset



8-bit (opcode == 000)

```
ST1 { <Vt>.B }[<index>], [<Xn|SP>]
```

16-bit (opcode == 010 && size == x0)

```
ST1 { <Vt>.H }[<index>], [<Xn|SP>]
```

32-bit (opcode == 100 && size == 00)

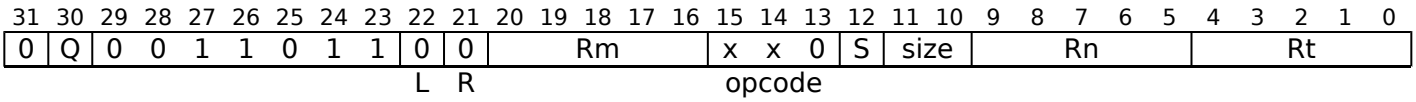
```
ST1 { <Vt>.S }[<index>], [<Xn|SP>]
```

64-bit (opcode == 100 && S == 0 && size == 01)

```
ST1 { <Vt>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index



8-bit, immediate offset (Rm == 11111 && opcode == 000)

```
ST1 { <Vt>.B }[<index>], [<Xn|SP>], #1
```

8-bit, register offset (Rm != 11111 && opcode == 000)

```
ST1 { <Vt>.B }[<index>], [<Xn|SP>], <Xm>
```

16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)

```
ST1 { <Vt>.H }[<index>], [<Xn|SP>], #2
```

16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)

```
ST1 { <Vt>.H }[<index>], [<Xn|SP>], <Xm>
```

32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)

```
ST1 { <Vt>.S }[<index>], [<Xn|SP>], #4
```

32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)

```
ST1 { <Vt>.S }[<index>], [<Xn|SP>], <Xm>
```

64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)

```
ST1 { <Vt>.D }[<index>], [<Xn|SP>], #8
```

64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)

```
ST1 { <Vt>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t, 128] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

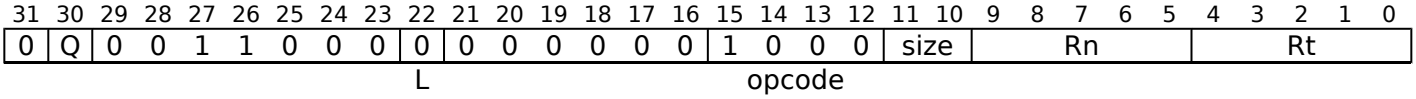
ST2 (multiple structures)

Store multiple 2-element structures from two registers. This instruction stores multiple 2-element structures from two SIMD&FP registers to memory, with interleaving. Every element of each register is stored.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

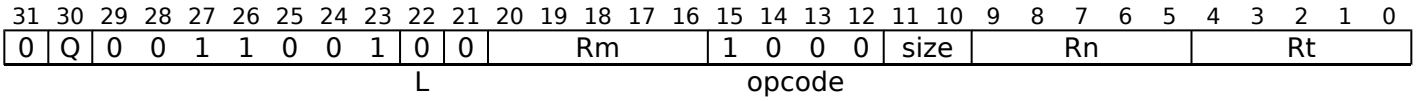
No offset



ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index



Immediate offset (Rm == 11111)

```
ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <imm>
```

Register offset (Rm != 11111)

```
ST2 { <Vt>.<T>, <Vt2>.<T> }, [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#16
1	#32

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;    // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
    otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt, datasize];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt, datasize] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

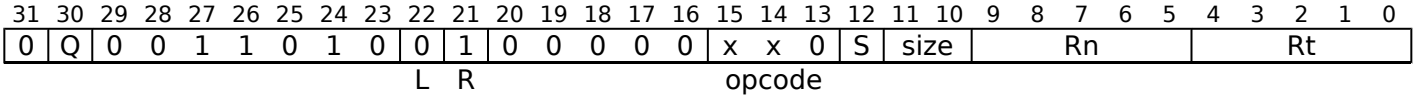
ST2 (single structure)

Store single 2-element structure from one lane of two registers. This instruction stores a 2-element structure to memory from corresponding elements of two SIMD&FP registers.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset



8-bit (opcode == 000)

```
ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>]
```

16-bit (opcode == 010 && size == x0)

```
ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>]
```

32-bit (opcode == 100 && size == 00)

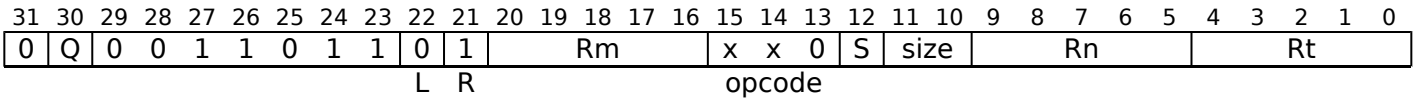
```
ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>]
```

64-bit (opcode == 100 && S == 0 && size == 01)

```
ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index



8-bit, immediate offset (Rm == 11111 && opcode == 000)

```
ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], #2
```

8-bit, register offset (Rm != 11111 && opcode == 000)

```
ST2 { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], <Xm>
```

16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)

```
ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], #4
```

16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)

```
ST2 { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], <Xm>
```

32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)

```
ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], #8
```

32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)

```
ST2 { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], <Xm>
```

64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)

```
ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], #16
```

64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)

```
ST2 { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t, 128] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

Operational information

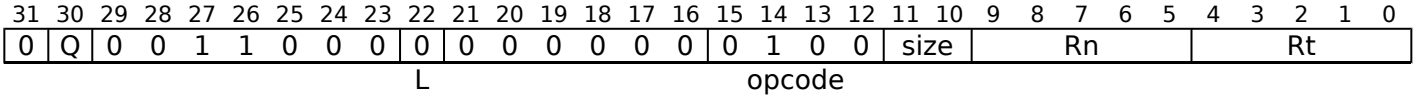
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST3 (multiple structures)

Store multiple 3-element structures from three registers. This instruction stores multiple 3-element structures to memory from three SIMD&FP registers, with interleaving. Every element of each register is stored.
Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.
It has encodings from 2 classes: No offset and Post-index

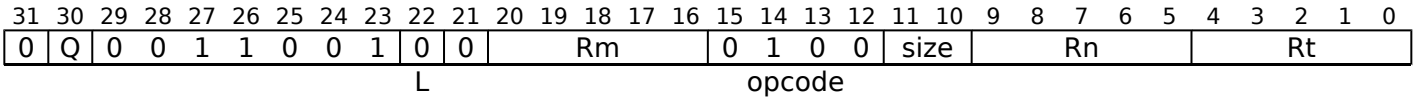
No offset



ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index



Immediate offset (Rm == 11111)

ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <imm>

Register offset (Rm != 11111)

ST3 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T> }, [<Xn|SP>], <Xm>

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#24
1	#48

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;    // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
    otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt, datasize];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt, datasize] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

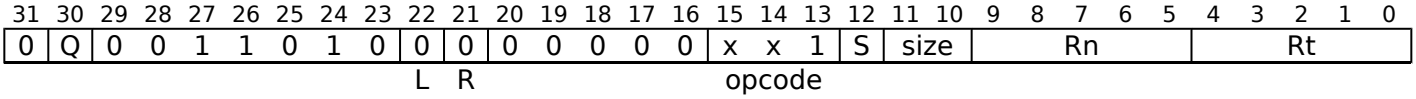
ST3 (single structure)

Store single 3-element structure from one lane of three registers. This instruction stores a 3-element structure to memory from corresponding elements of three SIMD&FP registers.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset



8-bit (opcode == 001)

```
ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>]
```

16-bit (opcode == 011 && size == x0)

```
ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>]
```

32-bit (opcode == 101 && size == 00)

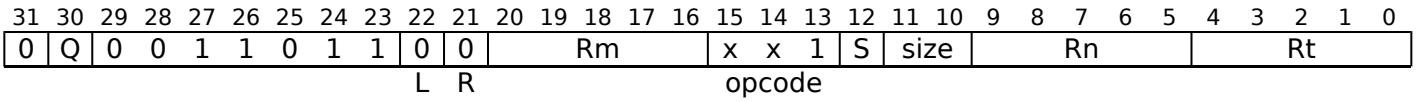
```
ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>]
```

64-bit (opcode == 101 && S == 0 && size == 01)

```
ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index



8-bit, immediate offset (Rm == 11111 && opcode == 001)

```
ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], #3
```

8-bit, register offset (Rm != 11111 && opcode == 001)

```
ST3 { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], <Xm>
```

16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

```
ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], #6
```

16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

```
ST3 { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], <Xm>
```

32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

```
ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], #12
```

32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

```
ST3 { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], <Xm>
```

64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

```
ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], #24
```

64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

```
ST3 { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3>	Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t, 128] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

Operational information

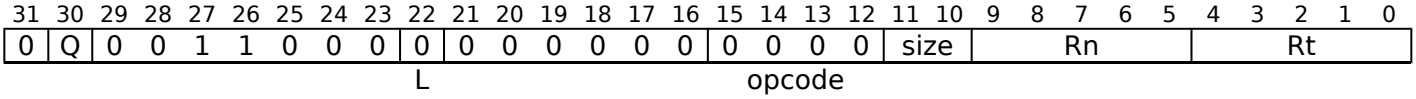
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST4 (multiple structures)

Store multiple 4-element structures from four registers. This instruction stores multiple 4-element structures to memory from four SIMD&FP registers, with interleaving. Every element of each register is stored.
Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.
It has encodings from 2 classes: No offset and Post-index

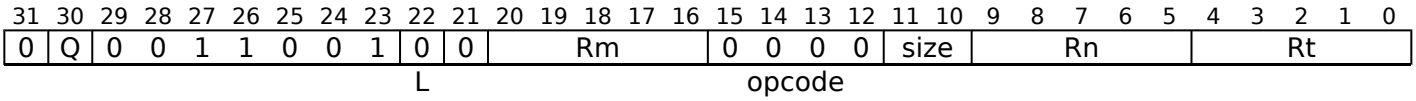
No offset



ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>]

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index



Immediate offset (Rm == 11111)

ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <imm>

Register offset (Rm != 11111)

ST4 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T> }, [<Xn|SP>], <Xm>

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in “Q”:

Q	<imm>
0	#32
1	#64

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;    // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;    // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;    // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;    // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;    // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;    // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;    // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;    // LD/ST1 (2 registers)
    otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt, datasize];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address+offs, ebytes, AccType_VEC];
                V[tt, datasize] = rval;
            else // memop == MemOp_STORE
                Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

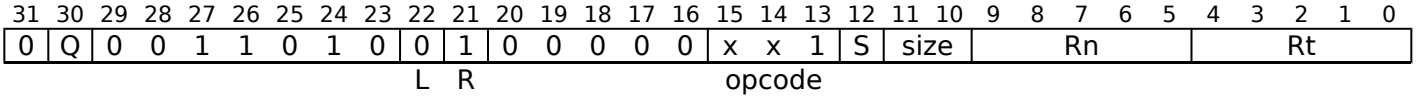
ST4 (single structure)

Store single 4-element structure from one lane of four registers. This instruction stores a 4-element structure to memory from corresponding elements of four SIMD&FP registers.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset



8-bit (opcode == 001)

```
ST4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>]
```

16-bit (opcode == 011 && size == x0)

```
ST4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>]
```

32-bit (opcode == 101 && size == 00)

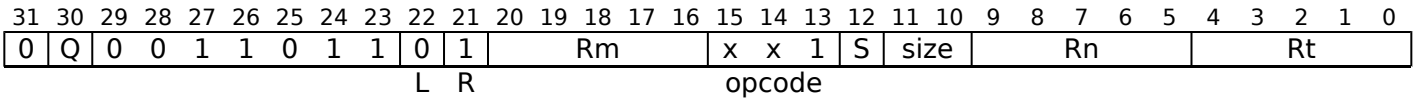
```
ST4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>]
```

64-bit (opcode == 101 && S == 0 && size == 01)

```
ST4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>]
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
boolean tag_checked = wback || n != 31;
```

Post-index



8-bit, immediate offset (Rm == 11111 && opcode == 001)

```
ST4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], #4
```

8-bit, register offset (Rm != 11111 && opcode == 001)

```
ST4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], <Xm>
```

16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

```
ST4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], #8
```

16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

```
ST4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], <Xm>
```

32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

```
ST4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], #16
```

32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

```
ST4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], <Xm>
```

64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

```
ST4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], #32
```

64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

```
ST4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], <Xm>
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
boolean tag_checked = wback || n != 31;
```

Assembler Symbols

<Vt>	Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.
<Vt2>	Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.
<Vt3>	Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.
<Vt4>	Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.
<index>	For the 8-bit variant: is the element index, encoded in "Q:S:size". For the 16-bit variant: is the element index, encoded in "Q:S:size<1>". For the 32-bit variant: is the element index, encoded in "Q:S". For the 64-bit variant: is the element index, encoded in "Q".
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer init_scale = UInt(opcode<2:1>);
integer scale = init_scale;
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);    // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);    // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);    // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);    // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

offs = Zeros(64);
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address+offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t, datasize] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t, 128];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address+offs, ebytes, AccType_VEC];
            V[t, 128] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address+offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m, 64];
    if n == 31 then
        SP[] = address + offs;
    else
        X[n, 64] = address + offs;
```

Operational information

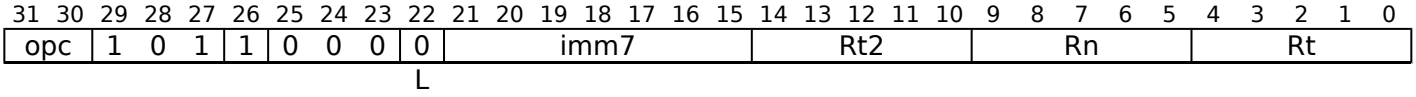
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STNP (SIMD&FP)

Store Pair of SIMD&FP registers, with Non-temporal hint. This instruction stores a pair of SIMD&FP registers to memory, issuing a hint to the memory system that the access is non-temporal. The address used for the store is calculated from an address from a base register value and an immediate offset. For information about non-temporal pair instructions, see *Load/Store SIMD and Floating-point Non-temporal pair*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



32-bit (opc == 00)

```
STNP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

64-bit (opc == 01)

```
STNP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

128-bit (opc == 10)

```
STNP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

```
// Empty.
```

Assembler Symbols

<Dt1>	Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt2>	Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Qt1>	Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt2>	Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<St1>	Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<St2>	Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4. For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8. For the 128-bit variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = n != 31;
```

Operation

```
CheckFPEnabled64();
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

data1 = V[t, datasize];
data2 = V[t2, datasize];
Mem[address, dbytes, AccType_VECSTREAM] = data1;
Mem[address+dbytes, dbytes, AccType_VECSTREAM] = data2;
```

Operational information

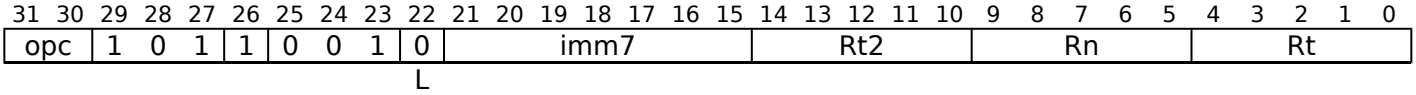
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STP (SIMD&FP)

Store Pair of SIMD&FP registers. This instruction stores a pair of SIMD&FP registers to memory. The address used for the store is calculated from a base register value and an immediate offset.
Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.
It has encodings from 3 classes: Post-index , Pre-index and Signed offset

Post-index



32-bit (opc == 00)

```
STP <St1>, <St2>, [<Xn|SP>], #<imm>
```

64-bit (opc == 01)

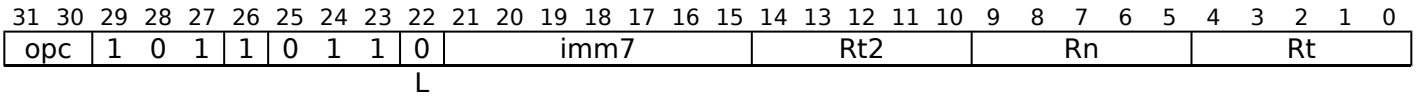
```
STP <Dt1>, <Dt2>, [<Xn|SP>], #<imm>
```

128-bit (opc == 10)

```
STP <Qt1>, <Qt2>, [<Xn|SP>], #<imm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
```

Pre-index



32-bit (opc == 00)

```
STP <St1>, <St2>, [<Xn|SP>, #<imm>]!
```

64-bit (opc == 01)

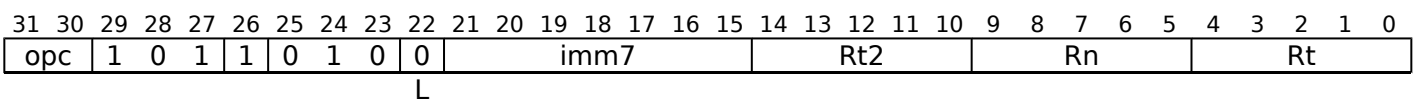
```
STP <Dt1>, <Dt2>, [<Xn|SP>, #<imm>]!
```

128-bit (opc == 10)

```
STP <Qt1>, <Qt2>, [<Xn|SP>, #<imm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
```

Signed offset



32-bit (opc == 00)

```
STP <St1>, <St2>, [<Xn|SP>{, #<imm>}]
```

64-bit (opc == 01)

```
STP <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}]
```

128-bit (opc == 10)

```
STP <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
```

Assembler Symbols

<Dt1>	Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt2>	Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Qt1>	Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt2>	Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<St1>	Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.
<St2>	Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4. For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4. For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8. For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8. For the 128-bit post-index and 128-bit pre-index variant: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field as <imm>/16. For the 128-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
boolean tag_checked = wback || n != 31;
```

Operation

```
CheckFPEnabled64();
bits(64) address;
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

data1 = V[t, datasize];
data2 = V[t2, datasize];
Mem[address, dbytes, AccType_VEC] = data1;
Mem[address+dbytes, dbytes, AccType_VEC] = data2;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

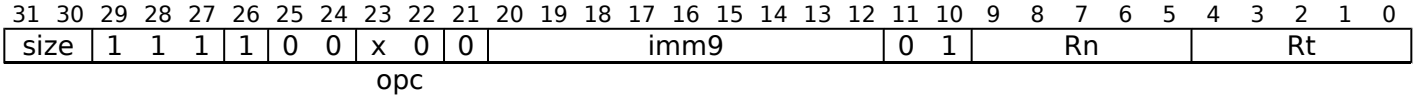
STR (immediate, SIMD&FP)

Store SIMD&FP register (immediate offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an immediate offset.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index



8-bit (size == 00 && opc == 00)

```
STR <Bt>, [<Xn|SP>], #<simm>
```

16-bit (size == 01 && opc == 00)

```
STR <Ht>, [<Xn|SP>], #<simm>
```

32-bit (size == 10 && opc == 00)

```
STR <St>, [<Xn|SP>], #<simm>
```

64-bit (size == 11 && opc == 00)

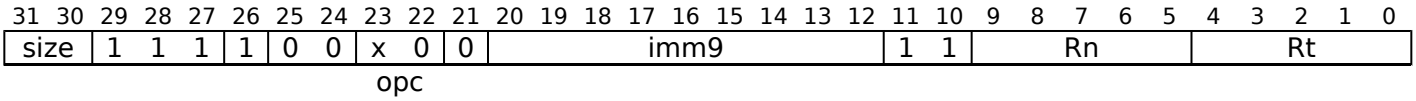
```
STR <Dt>, [<Xn|SP>], #<simm>
```

128-bit (size == 00 && opc == 10)

```
STR <Qt>, [<Xn|SP>], #<simm>
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index



8-bit (size == 00 && opc == 00)

```
STR <Bt>, [<Xn|SP>, #<simm>]!
```

16-bit (size == 01 && opc == 00)

```
STR <Ht>, [<Xn|SP>, #<simm>]!
```

32-bit (size == 10 && opc == 00)

```
STR <St>, [<Xn|SP>, #<simm>]!
```

64-bit (size == 11 && opc == 00)

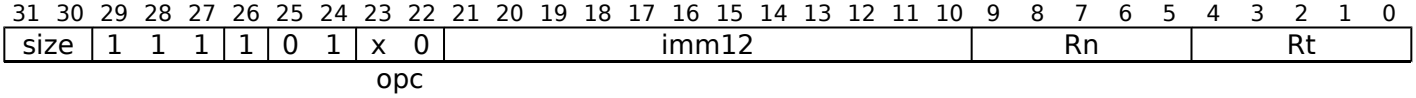
```
STR <Dt>, [<Xn|SP>, #<simm>]!
```

128-bit (size == 00 && opc == 10)

```
STR <Qt>, [<Xn|SP>, #<simm>]!
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset



8-bit (size == 00 && opc == 00)

```
STR <Bt>, [<Xn|SP>{, #<pimm>}]
```

16-bit (size == 01 && opc == 00)

```
STR <Ht>, [<Xn|SP>{, #<pimm>}]
```

32-bit (size == 10 && opc == 00)

```
STR <St>, [<Xn|SP>{, #<pimm>}]
```

64-bit (size == 11 && opc == 00)

```
STR <Dt>, [<Xn|SP>{, #<pimm>}]
```

128-bit (size == 00 && opc == 10)

```
STR <Qt>, [<Xn|SP>{, #<pimm>}]
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

Assembler Symbols

<Bt>	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht>	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.
<pimm>	For the 8-bit variant: is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field. For the 16-bit variant: is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2. For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4. For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8. For the 128-bit variant: is the optional positive immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0 and encoded in the "imm12" field as <pimm>/16.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (wback || n != 31);
```

Operation

```
CheckFPEnabled64();
bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

if !postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t, datasize];
        Mem[address, datasize DIV 8, AccType_VEC] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, AccType_VEC];
        V[t, datasize] = data;

if wback then
    if postindex then
        address = address + offset;
    if n == 31 then
        SP[] = address;
    else
        X[n, 64] = address;
```

Operational information

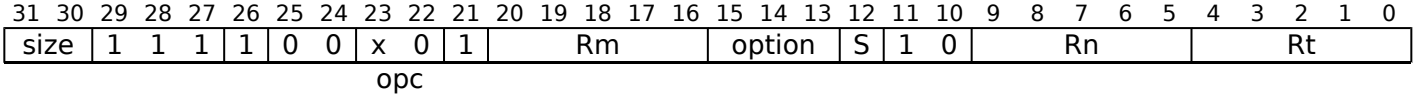
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STR (register, SIMD&FP)

Store SIMD&FP register (register offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an offset register value. The offset can be optionally shifted and extended.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



8-bit (size == 00 && opc == 00 && option != 011)

```
STR <Bt>, [<Xn|SP>, (<Wm>|<Xm>), <extend> {<amount>}]
```

8-bit (size == 00 && opc == 00 && option == 011)

```
STR <Bt>, [<Xn|SP>, <Xm>{, LSL <amount>}]
```

16-bit (size == 01 && opc == 00)

```
STR <Ht>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

32-bit (size == 10 && opc == 00)

```
STR <St>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

64-bit (size == 11 && opc == 00)

```
STR <Dt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

128-bit (size == 00 && opc == 10)

```
STR <Qt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend> {<amount>}}]
```

```
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
if option<1> == '0' then UNDEFINED;    // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

<Bt>	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht>	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Wm>	When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.
<Xm>	When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> For the 8-bit variant: is the index extend specifier, encoded in “option”:

option	<extend>
010	UXTW
110	SXTW
111	SXTX

For the 128-bit, 16-bit, 32-bit and 64-bit variant: is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in “option”:

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount> For the 8-bit variant: is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

For the 16-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#1

For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#2

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#3

For the 128-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in “S”:

S	<amount>
0	#0
1	#4

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift, 64);
CheckFPEnabled64();
bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t, datasize];
        Mem[address, datasize DIV 8, AccType_VEC] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, AccType_VEC];
        V[t, datasize] = data;
```

Operational information

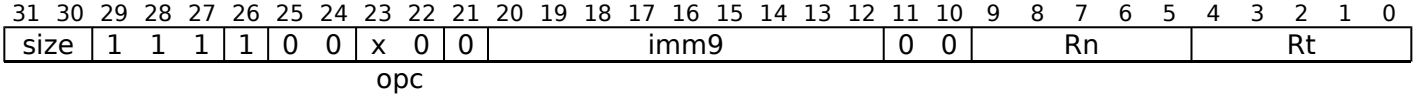
If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STUR (SIMD&FP)

Store SIMD&FP register (unscaled offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an optional immediate offset.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



8-bit (size == 00 && opc == 00)

```
STUR <Bt>, [<Xn|SP>{, #<simm>}]
```

16-bit (size == 01 && opc == 00)

```
STUR <Ht>, [<Xn|SP>{, #<simm>}]
```

32-bit (size == 10 && opc == 00)

```
STUR <St>, [<Xn|SP>{, #<simm>}]
```

64-bit (size == 11 && opc == 00)

```
STUR <Dt>, [<Xn|SP>{, #<simm>}]
```

128-bit (size == 00 && opc == 10)

```
STUR <Qt>, [<Xn|SP>{, #<simm>}]
```

```
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Bt>	Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Dt>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Ht>	Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Qt>	Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<St>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<simm>	Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
boolean tag_checked = memop != MemOp_PREFETCH && (n != 31);
```

Operation

```
CheckFPEnabled64();
bits(64) address;
bits(datasize) data;

if HaveMTE2Ext() then
    SetTagCheckedInstruction(tag_checked);

if n == 31 then
    CheckSPAlignment();
    address = SP[];
else
    address = X[n, 64];

address = address + offset;

case memop of
    when MemOp_STORE
        data = V[t, datasize];
        Mem[address, datasize DIV 8, AccType_VEC] = data;

    when MemOp_LOAD
        data = Mem[address, datasize DIV 8, AccType_VEC];
        V[t, datasize] = data;
```

Operational information

If PSTATE.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

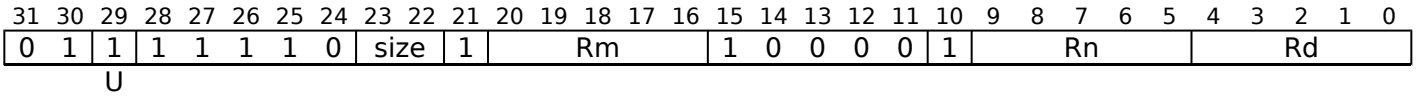
SUB (vector)

Subtract (vector). This instruction subtracts each vector element in the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

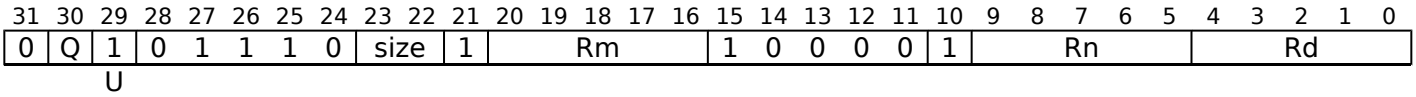
Scalar



SUB <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size != '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean sub_op = (U == '1');
```

Vector



SUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean sub_op = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize];
    element2 = Elem[operand2, e, esize];
    if sub_op then
        Elem[result, e, esize] = element1 - element2;
    else
        Elem[result, e, esize] = element1 + element2;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

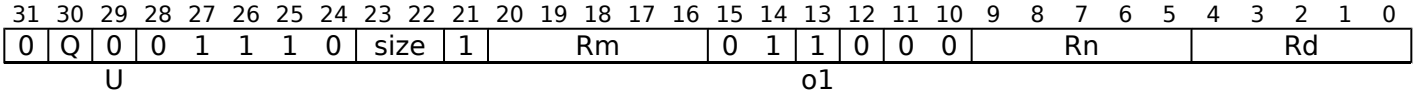
SUBHN, SUBHN2

Subtract returning High Narrow. This instruction subtracts each vector element in the second source SIMD&FP register from the corresponding vector element in the first source SIMD&FP register, places the most significant half of the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are signed integer values.

The results are truncated. For rounded results, see *RSUBHN*.

The SUBHN instruction writes the vector to the lower half of the destination register and clears the upper half, while the SUBHN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



SUBHN{2} <Vd>.<Tb>, <Vn>.<Ta>, <Vm>.<Ta>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean round = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n, 2*datasize];
bits(2*datasize) operand2 = V[m, 2*datasize];
bits(datasize) result;
integer round_const = if round then 1 << (esize - 1) else 0;
bits(2*esize) element1;
bits(2*esize) element2;
bits(2*esize) sum;

for e = 0 to elements-1
    element1 = Elem[operand1, e, 2*esize];
    element2 = Elem[operand2, e, 2*esize];
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    sum = sum + round_const;
    Elem[result, e, esize] = sum<2*esize-1:esize>;

Vpart[d, part, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

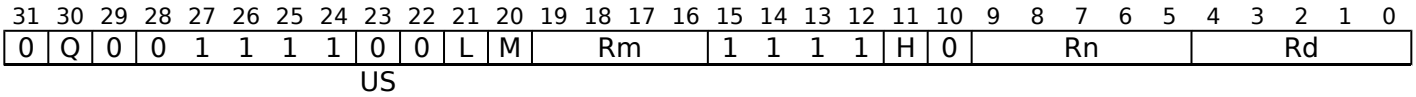
SUDOT (by element)

Dot product index form with signed and unsigned integers. This instruction performs the dot product of the four signed 8-bit integer values in each 32-bit element of the first source register with the four unsigned 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination vector.

From Armv8.2 to Armv8.5, this is an OPTIONAL instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. *ID_AA64ISAR1_EL1*.I8MM indicates whether this instruction is supported.

Vector

(FEAT_I8MM)



SUDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.4B[<index>]

```
if !HaveInt8MatMulExt() then UNDEFINED;
boolean op1_unsigned = (US == '1');
boolean op2_unsigned = (US == '0');
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer d = UInt(Rd);
integer i = UInt(H:L);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	8B
1	16B

<Vm> Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<index> Is the immediate index of a 32-bit group of four 8-bit values in the range 0 to 3, encoded in the "H:L" fields.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(128) operand2 = V[m, 128];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;

for e = 0 to elements-1
    bits(32) res = Elem[operand3, e, 32];
    for b = 0 to 3
        integer element1 = Int(Elem[operand1, 4*e+b, 8], op1_unsigned);
        integer element2 = Int(Elem[operand2, 4*i+b, 8], op2_unsigned);
        res = res + element1 * element2;
    Elem[result, e, 32] = res;
V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SUQADD

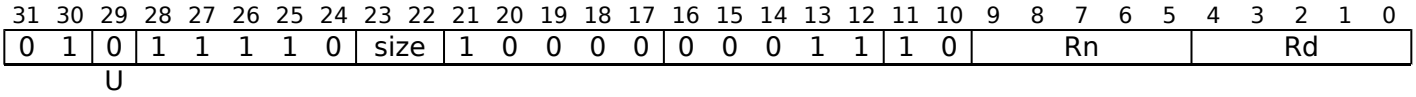
Signed saturating Accumulate of Unsigned value. This instruction adds the unsigned integer values of the vector elements in the source SIMD&FP register to corresponding signed integer values of the vector elements in the destination SIMD&FP register, and writes the resulting signed integer values to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

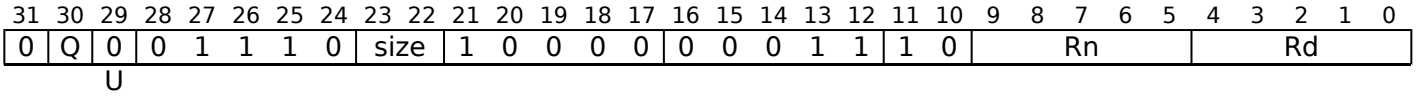
Scalar



SUQADD <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean unsigned = (U == '1');
```

Vector



SUQADD <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;

bits(datasize) operand2 = V[d, datasize];
integer op1;
integer op2;
boolean sat;

for e = 0 to elements-1
    op1 = Int(Elem[operand, e, esize], !unsigned);
    op2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], sat) = SatQ(op1 + op2, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SXTL, SXTL2

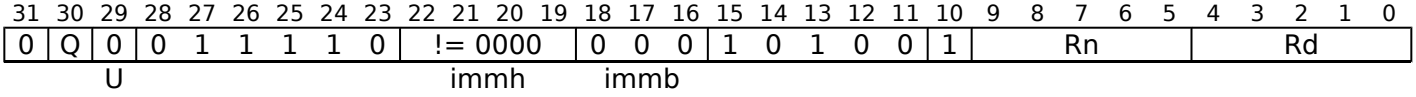
Signed extend Long. This instruction duplicates each vector element in the lower or upper half of the source SIMD&FP register into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are signed integer values.

The SXTL instruction extracts the source vector from the lower half of the source register. The SXTL2 instruction extracts the source vector from the upper half of the source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of SSHLL, SSHLL2. This means:

- The encodings in this description are named to match the encodings of SSHLL, SSHLL2.
- The description of SSHLL, SSHLL2 gives the operational pseudocode for this instruction.



SXTL{2} <Vd>.<Ta>, <Vn>.<Tb>

is equivalent to

SSHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #0

and is the preferred disassembly when `BitCount(immh) == 1`.

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

Operation

The description of SSHLL, SSHLL2 gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

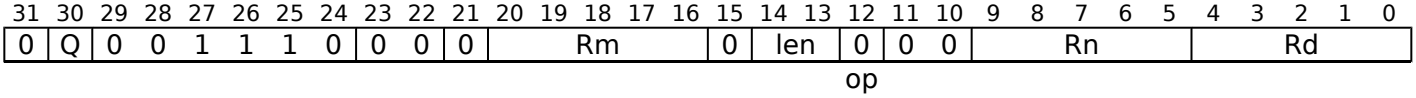
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

TBL

Table vector Lookup. This instruction reads each value from the vector elements in the index source SIMD&FP register, uses each result as an index to perform a lookup in a table of bytes that is described by one to four source table SIMD&FP registers, places the lookup result in a vector, and writes the vector to the destination SIMD&FP register. If an index is out of range for the table, the result for that lookup is 0. If more than one source register is used to describe the table, the first source register describes the lowest bytes of the table.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Two register table (len == 01)

```
TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B }, <Vm>.<Ta>
```

Three register table (len == 10)

```
TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B }, <Vm>.<Ta>
```

Four register table (len == 11)

```
TBL <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B }, <Vm>.<Ta>
```

Single register table (len == 00)

```
TBL <Vd>.<Ta>, { <Vn>.16B }, <Vm>.<Ta>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
integer regs = UInt(len) + 1;
boolean is_tbl = (op == '0');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	8B
1	16B

<Vn> For the four register table, three register table and two register table variant: is the name of the first SIMD&FP table register, encoded in the "Rn" field.

For the single register table variant: is the name of the SIMD&FP table register, encoded in the "Rn" field.

<Vn+1> Is the name of the second SIMD&FP table register, encoded as "Rn" plus 1 modulo 32.

<Vn+2> Is the name of the third SIMD&FP table register, encoded as "Rn" plus 2 modulo 32.

<Vn+3> Is the name of the fourth SIMD&FP table register, encoded as "Rn" plus 3 modulo 32.

<Vm> Is the name of the SIMD&FP index register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) indices = V[m, datasize];
bits(128*regs) table = Zeros(128 * regs);
bits(datasize) result;
integer index;

// Create table from registers
for i = 0 to regs-1
    table<128*i+127:128*i> = V[n, 128];
    n = (n + 1) MOD 32;

result = if is_tbl then Zeros(datasize) else V[d, datasize];
for i = 0 to elements-1
    index = UInt(Elem[indices, i, 8]);
    if index < 16 * regs then
        Elem[result, i, 8] = Elem[table, index, 8];

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

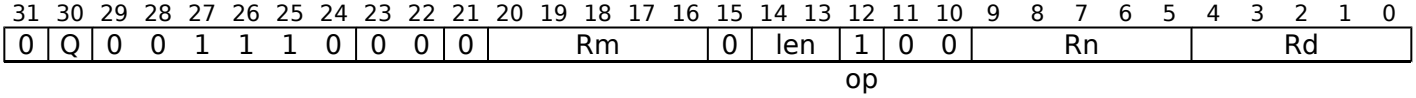
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

TBX

Table vector lookup extension. This instruction reads each value from the vector elements in the index source SIMD&FP register, uses each result as an index to perform a lookup in a table of bytes that is described by one to four source table SIMD&FP registers, places the lookup result in a vector, and writes the vector to the destination SIMD&FP register. If an index is out of range for the table, the existing value in the vector element of the destination register is left unchanged. If more than one source register is used to describe the table, the first source register describes the lowest bytes of the table.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



Two register table (len == 01)

```
TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B }, <Vm>.<Ta>
```

Three register table (len == 10)

```
TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B }, <Vm>.<Ta>
```

Four register table (len == 11)

```
TBX <Vd>.<Ta>, { <Vn>.16B, <Vn+1>.16B, <Vn+2>.16B, <Vn+3>.16B }, <Vm>.<Ta>
```

Single register table (len == 00)

```
TBX <Vd>.<Ta>, { <Vn>.16B }, <Vm>.<Ta>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 8;
integer regs = UInt(len) + 1;
boolean is_tbl = (op == '0');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	8B
1	16B

<Vn> For the four register table, three register table and two register table variant: is the name of the first SIMD&FP table register, encoded in the "Rn" field.

For the single register table variant: is the name of the SIMD&FP table register, encoded in the "Rn" field.

<Vn+1> Is the name of the second SIMD&FP table register, encoded as "Rn" plus 1 modulo 32.

<Vn+2> Is the name of the third SIMD&FP table register, encoded as "Rn" plus 2 modulo 32.

<Vn+3> Is the name of the fourth SIMD&FP table register, encoded as "Rn" plus 3 modulo 32.

<Vm> Is the name of the SIMD&FP index register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) indices = V[m, datasize];
bits(128*regs) table = Zeros(128 * regs);
bits(datasize) result;
integer index;

// Create table from registers
for i = 0 to regs-1
    table<128*i+127:128*i> = V[n, 128];
    n = (n + 1) MOD 32;

result = if is_tbl then Zeros(datasize) else V[d, datasize];
for i = 0 to elements-1
    index = UInt(Elem[indices, i, 8]);
    if index < 16 * regs then
        Elem[result, i, 8] = Elem[table, index, 8];

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

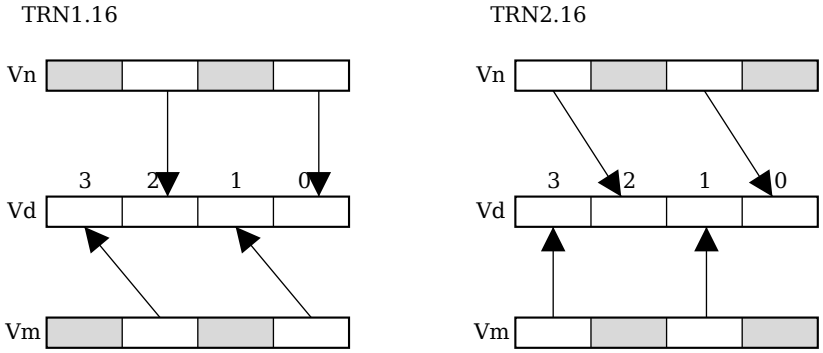
TRN1

Transpose vectors (primary). This instruction reads corresponding even-numbered vector elements from the two source SIMD&FP registers, starting at zero, places each result into consecutive elements of a vector, and writes the vector to the destination SIMD&FP register. Vector elements from the first source register are placed into even-numbered elements of the destination vector, starting at zero, while vector elements from the second source register are placed into odd-numbered elements of the destination vector.

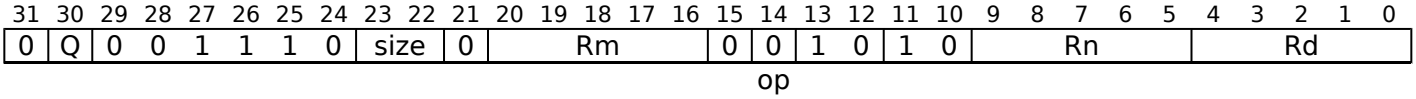
Note

By using this instruction with TRN2, a 2 x 2 matrix can be transposed.

The following figure shows an example of the operation of TRN1 and TRN2 halfword operations where Q = 0.



Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



TRN1 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, 2*p+part, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, 2*p+part, esize];

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

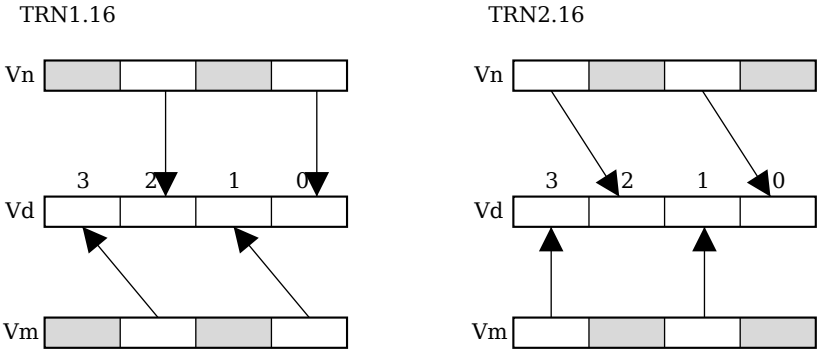
TRN2

Transpose vectors (secondary). This instruction reads corresponding odd-numbered vector elements from the two source SIMD&FP registers, places each result into consecutive elements of a vector, and writes the vector to the destination SIMD&FP register. Vector elements from the first source register are placed into even-numbered elements of the destination vector, starting at zero, while vector elements from the second source register are placed into odd-numbered elements of the destination vector.

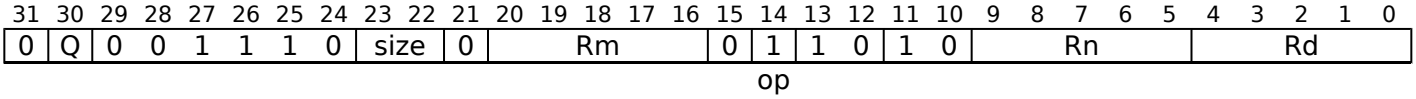
Note

By using this instruction with TRN1, a 2 x 2 matrix can be transposed.

The following figure shows an example of the operation of TRN1 and TRN2 halfword operations where Q = 0.



Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



TRN2 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, 2*p+part, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, 2*p+part, esize];

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

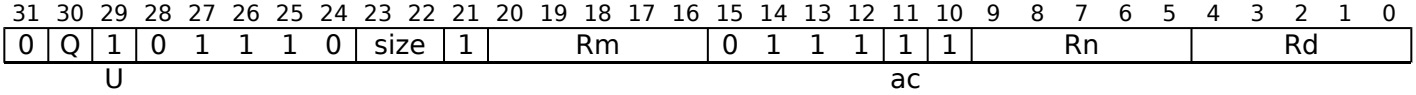
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UABA

Unsigned Absolute difference and Accumulate. This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the elements of the vector of the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UABA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

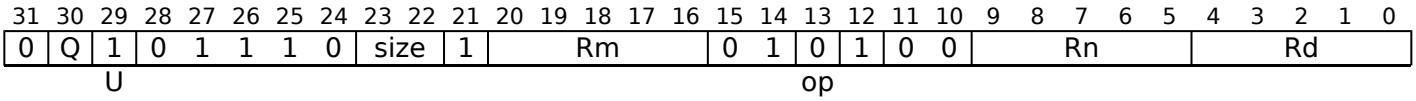
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UABAL, UABAL2

Unsigned Absolute difference and Accumulate Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, and accumulates the absolute values of the results into the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The UABAL instruction extracts each source vector from the lower half of each source register. The UABAL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UABAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d, 2*datasize] else Zeros(2 * datasize);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d, 2*datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

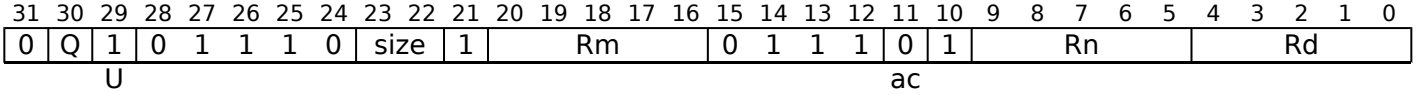
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UABD

Unsigned Absolute Difference (vector). This instruction subtracts the elements of the vector of the second source SIMD&FP register from the corresponding elements of the first source SIMD&FP register, places the absolute values of the results into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UABD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean accumulate = (ac == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
bits(esize) absdiff;

result = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;
V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

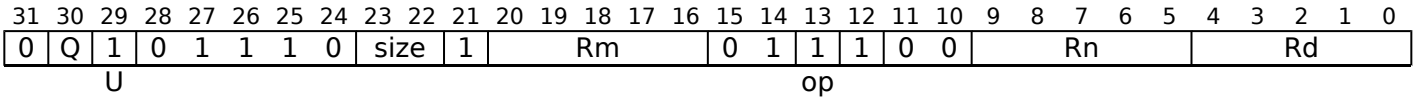
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UABDL, UABDL2

Unsigned Absolute Difference Long. This instruction subtracts the vector elements in the lower or upper half of the second source SIMD&FP register from the corresponding vector elements of the first source SIMD&FP register, places the absolute value of the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The UABDL instruction extracts each source vector from the lower half of each source register. The UABDL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UABDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean accumulate = (op == '0');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) absdiff;

result = if accumulate then V[d, 2*datasize] else Zeros(2 * datasize);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    absdiff = Abs(element1-element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + absdiff;
V[d, 2*datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

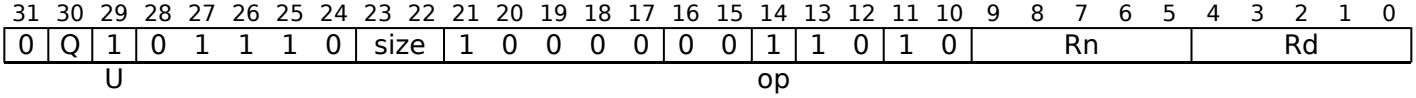
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UADALP

Unsigned Add and Accumulate Long Pairwise. This instruction adds pairs of adjacent unsigned integer values from the vector in the source SIMD&FP register and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UADALP <Vd>.<Ta>, <Vn>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

if acc then result = V[d, datasize];
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>;
    if acc then
        Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
    else
        Elem[result, e, 2*esize] = sum;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

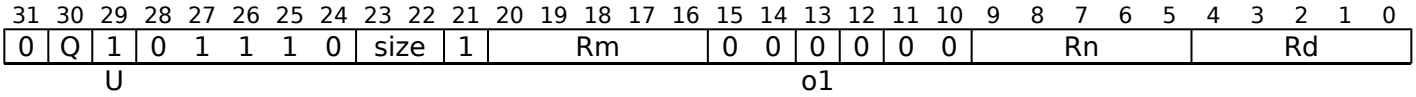
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UADDL, UADDL2

Unsigned Add Long (vector). This instruction adds each vector element in the lower or upper half of the first source SIMD&FP register to the corresponding vector element of the second source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

The UADDL instruction extracts each source vector from the lower half of each source register. The UADDL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UADDL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d, 2*datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

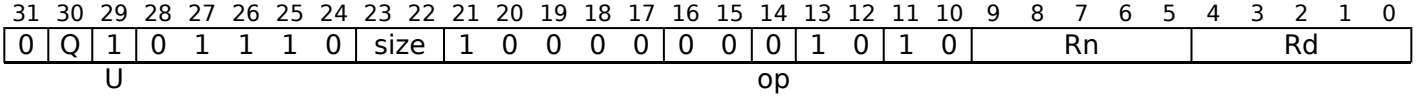
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UADDLP

Unsigned Add Long Pairwise. This instruction adds pairs of adjacent unsigned integer values from the vector in the source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UADDLP <Vd>.<Ta>, <Vn>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV (2 * esize);
boolean acc = (op == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Ta>
00	0	4H
00	1	8H
01	0	2S
01	1	4S
10	0	1D
10	1	2D
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;

bits(2*esize) sum;
integer op1;
integer op2;

if acc then result = V[d, datasize];
for e = 0 to elements-1
    op1 = Int(Elem[operand, 2*e+0, esize], unsigned);
    op2 = Int(Elem[operand, 2*e+1, esize], unsigned);
    sum = (op1+op2)<2*esize-1:0>;
    if acc then
        Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + sum;
    else
        Elem[result, e, 2*esize] = sum;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

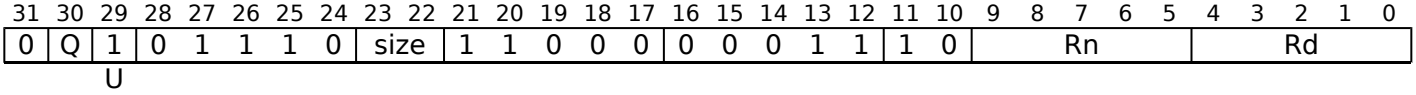
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UADDLV

Unsigned sum Long across Vector. This instruction adds every vector element in the source SIMD&FP register together, and writes the scalar result to the destination SIMD&FP register. The destination scalar is twice as long as the source vector elements. All the values in this instruction are unsigned integer values.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UADDLV <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

size	<V>
00	H
01	S
10	D
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
integer sum;

sum = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    sum = sum + Int(Elem[operand, e, esize], unsigned);

V[d, 2*esize] = sum<2*esize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

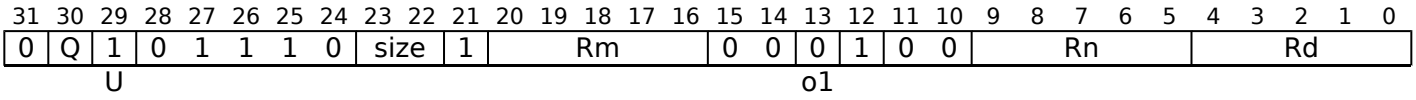
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UADDW, UADDW2

Unsigned Add Wide. This instruction adds the vector elements of the first source SIMD&FP register to the corresponding vector elements in the lower or upper half of the second source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. The vector elements of the destination register and the first source register are twice as long as the vector elements of the second source register. All the values in this instruction are unsigned integer values.

The UADDW instruction extracts vector elements from the lower half of the second source register. The UADDW2 instruction extracts vector elements from the upper half of the second source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UADDW{2} <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n, 2*datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, 2*esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d, 2*datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

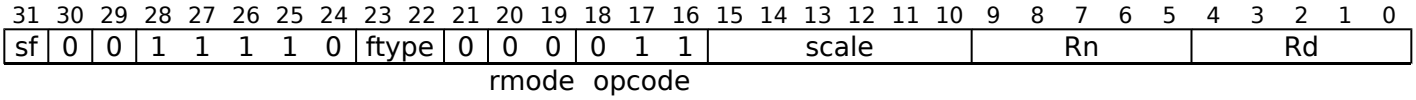
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UCVTF (scalar, fixed-point)

Unsigned fixed-point Convert to Floating-point (scalar). This instruction converts the unsigned value in the 32-bit or 64-bit general-purpose source register to a floating-point value using the rounding mode that is specified by the *FPCR*, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.



32-bit to half-precision (sf == 0 && ftype == 11)
(FEAT_FP16)

```
UCVTF <Hd>, <Wn>, #<fbits>
```

32-bit to single-precision (sf == 0 && ftype == 00)

```
UCVTF <Sd>, <Wn>, #<fbits>
```

32-bit to double-precision (sf == 0 && ftype == 01)

```
UCVTF <Dd>, <Wn>, #<fbits>
```

64-bit to half-precision (sf == 1 && ftype == 11)
(FEAT_FP16)

```
UCVTF <Hd>, <Xn>, #<fbits>
```

64-bit to single-precision (sf == 1 && ftype == 00)

```
UCVTF <Sd>, <Xn>, #<fbits>
```

64-bit to double-precision (sf == 1 && ftype == 01)

```
UCVTF <Dd>, <Xn>, #<fbits>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case ftype of
    when '00' fltsize = 32;
    when '01' fltsize = 64;
    when '10' UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

if sf == '0' && scale<5> == '0' then UNDEFINED;
integer fracbits = 64 - UInt(scale);

rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.
<fbits>	For the 32-bit to double-precision, 32-bit to half-precision and 32-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 32, encoded as 64 minus "scale". For the 64-bit to double-precision, 64-bit to half-precision and 64-bit to single-precision variant: is the number of bits after the binary point in the fixed-point source, in the range 1 to 64, encoded as 64 minus "scale".

Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
integer fsize = if merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;

intval = X[n, intsize];
fltval = if merge then V[d, fsize] else Zeros(fsize);
Elem[fltval, 0, fltsize] = FixedToFP(intval, fracbits, TRUE, fpcr, rounding, fltsize);
V[d, fsize] = fltval;
```

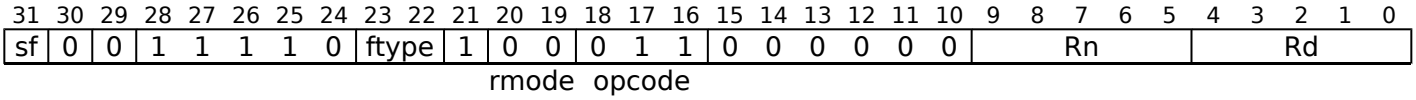
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UCVTF (scalar, integer)

Unsigned integer Convert to Floating-point (scalar). This instruction converts the unsigned integer value in the general-purpose source register to a floating-point value using the rounding mode that is specified by the *FPCR*, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



32-bit to half-precision (sf == 0 && ftype == 11)
(FEAT_FP16)

```
UCVTF <Hd>, <Wn>
```

32-bit to single-precision (sf == 0 && ftype == 00)

```
UCVTF <Sd>, <Wn>
```

32-bit to double-precision (sf == 0 && ftype == 01)

```
UCVTF <Dd>, <Wn>
```

64-bit to half-precision (sf == 1 && ftype == 11)
(FEAT_FP16)

```
UCVTF <Hd>, <Xn>
```

64-bit to single-precision (sf == 1 && ftype == 00)

```
UCVTF <Sd>, <Xn>
```

64-bit to double-precision (sf == 1 && ftype == 01)

```
UCVTF <Dd>, <Xn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer intsize = if sf == '1' then 64 else 32;
integer fltsize;
FPRounding rounding;

case ftype of
    when '00'
        fltsize = 32;
    when '01'
        fltsize = 64;
    when '10'
        UNDEFINED;
    when '11'
        if HaveFP16Ext() then
            fltsize = 16;
        else
            UNDEFINED;

rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Hd>	Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Rd" field.
<Xn>	Is the 64-bit name of the general-purpose source register, encoded in the "Rn" field.
<Wn>	Is the 32-bit name of the general-purpose source register, encoded in the "Rn" field.

Operation

```
CheckFPEnabled64();

FPCRType fpcr = FPCR[];
boolean merge = IsMerging(fpcr);
integer fsize = if merge then 128 else fltsize;
bits(fsize) fltval;
bits(intsize) intval;

intval = X[n, intsize];
fltval = if merge then V[d, fsize] else Zeros(fsize);
Elem[fltval, 0, fltsize] = FixedToFP(intval, 0, TRUE, fpcr, rounding, fltsize);
V[d, fsize] = fltval;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UCVTF (vector, fixed-point)

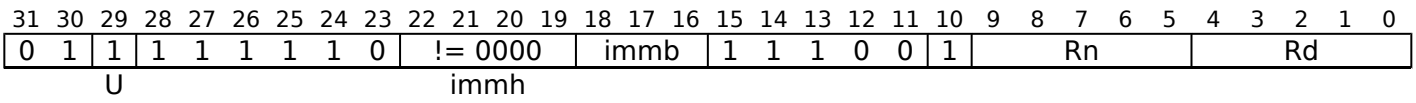
Unsigned fixed-point Convert to Floating-point (vector). This instruction converts each element in a vector from fixed-point to floating-point using the rounding mode that is specified by the *FPCR*, and writes the result to the SIMD&FP destination register.

A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

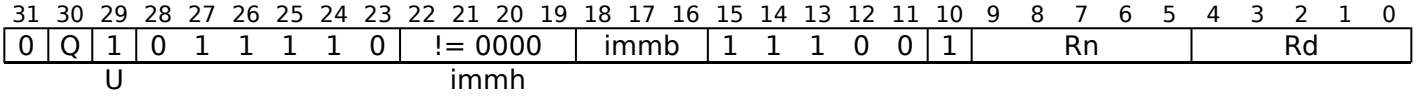
Scalar



UCVTF <V><d>, <V><n>, #<fbits>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh IN {'000x'} || (immh IN {'001x'} && !HaveFP16Ext()) then UNDEFINED;
integer esize = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
integer datasize = esize;
integer elements = 1;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR[]);
```

Vector



UCVTF <Vd>.<T>, <Vn>.<T>, #<fbits>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh IN {'000x'} || (immh IN {'001x'} && !HaveFP16Ext()) then UNDEFINED;
if immh<3>:Q == '10' then UNDEFINED;
integer esize = if immh IN {'1xxx'} then 64 else if immh IN {'01xx'} then 32 else 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer fracbits = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
FPRounding rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
000x	RESERVED
001x	H
01xx	S
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	x	RESERVED
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<fbits> For the scalar variant: is the number of fractional bits, in the range 1 to the operand width, encoded in “immh:immb”:

immh	<fbits>
000x	RESERVED
001x	(32-Uint(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

For the vector variant: is the number of fractional bits, in the range 1 to the element width, encoded in “immh:immb”:

immh	<fbits>
0000	SEE Advanced SIMD modified immediate
0001	RESERVED
001x	(32-Uint(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

bits(esize) element;
FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, fracbits, unsigned, fpcr, rounding, esize);

V[d, 128] = result;
```

UCVTF (vector, integer)

Unsigned integer Convert to Floating-point (vector). This instruction converts each element in a vector from an unsigned integer value to a floating-point value using the rounding mode that is specified by the *FPCR*, and writes the result to the SIMD&FP destination register.

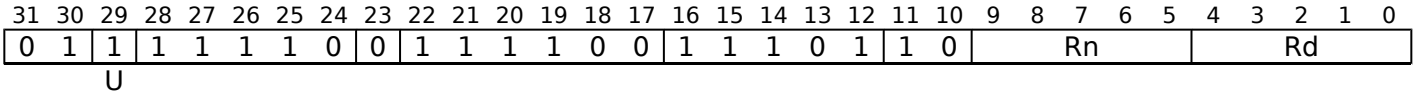
A floating-point exception can be generated by this instruction. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR*, or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the Security state and Exception level in which the instruction is executed, an attempt to execute the instruction might be trapped.

It has encodings from 4 classes: Scalar half precision , Scalar single-precision and double-precision , Vector half precision and Vector single-precision and double-precision

Scalar half precision

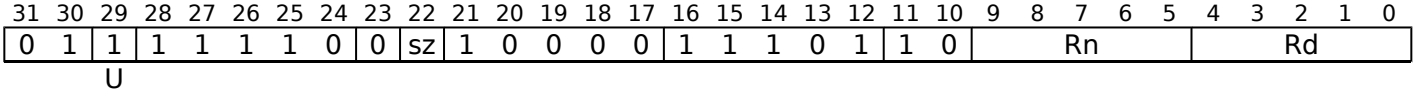
(FEAT_FP16)



UCVTF <Hd>, <Hn>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Scalar single-precision and double-precision

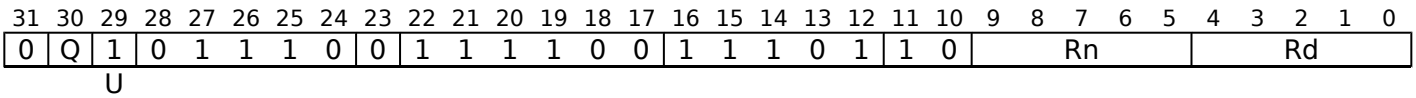


UCVTF <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 32 << UInt(sz);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Vector half precision

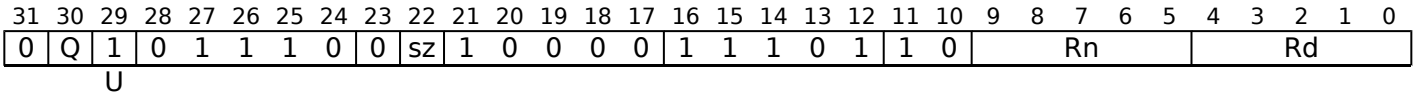
(FEAT_FP16)



UCVTF <Vd>.<T>, <Vn>.<T>

```
if !HaveFP16Ext() then UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 16;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Vector single-precision and double-precision



UCVTF <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz:Q == '10' then UNDEFINED;
integer esize = 32 << UInt(sz);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<Hd> Is the 16-bit name of the SIMD&FP destination register, encoded in the "Rd" field.

<Hn> Is the 16-bit name of the SIMD&FP source register, encoded in the "Rn" field.

<V> Is a width specifier, encoded in “sz”:

sz	<V>
0	S
1	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> For the half-precision variant: is an arrangement specifier, encoded in “Q”:

Q	<T>
0	4H
1	8H

For the single-precision and double-precision variant: is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	0	RESERVED
1	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];

FPCRType fpcr = FPCR[];
boolean merge = elements == 1 && IsMerging(fpcr);
bits(128) result = if merge then V[d, 128] else Zeros(128);

FPRounding rounding = FPRoundingMode(fpcr);
bits(esize) element;
for e = 0 to elements-1
    element = Elem[operand, e, esize];
    Elem[result, e, esize] = FixedToFP(element, 0, unsigned, fpcr, rounding, esize);

V[d, 128] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

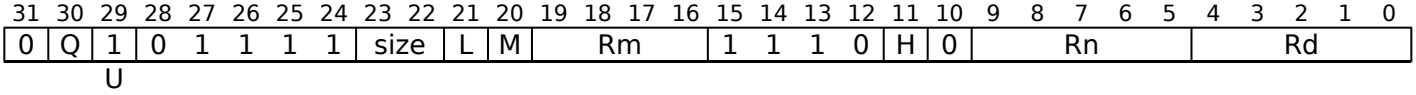
UDOT (by element)

Dot Product unsigned arithmetic (vector, by element). This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of an indexed 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.
Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.
In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_AA64ISAR0_EL1.DP indicates whether this instruction is supported.

Vector
(FEAT_DotProd)



UDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.4B[<index>]

```
if !HaveDOTPExt() then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U == '0');

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer index = UInt(H:L);

integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	8B
1	16B

<Vm> Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<index> Is the element index, encoded in the "H:L" fields.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(128) operand2 = V[m, 128];
bits(datasize) result = V[d, datasize];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        if signed then
            element1 = SInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4*index+i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4*index+i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UDOT (vector)

Dot Product unsigned arithmetic (vector). This instruction performs the dot product of the four unsigned 8-bit elements in each 32-bit element of the first source register with the four unsigned 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

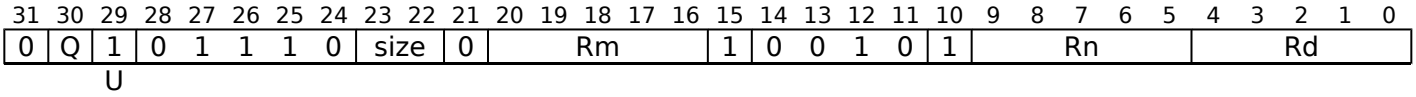
Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

ID_AA64ISAR0_EL1.DP indicates whether this instruction is supported.

Vector
(FEAT_DotProd)



UDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
if !HaveDOTPExt() then UNDEFINED;
if size != '10' then UNDEFINED;
boolean signed = (U == '0');
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	8B
1	16B

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

result = V[d, datasize];
for e = 0 to elements-1
    integer res = 0;
    integer element1, element2;
    for i = 0 to 3
        if signed then
            element1 = SInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = SInt(Elem[operand2, 4*e+i, esize DIV 4]);
        else
            element1 = UInt(Elem[operand1, 4*e+i, esize DIV 4]);
            element2 = UInt(Elem[operand2, 4*e+i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = Elem[result, e, esize] + res;
V[d, datasize] = result;
```

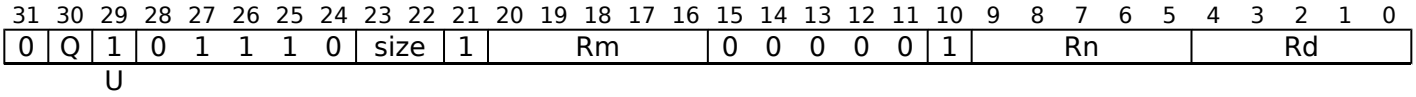
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UHADD

Unsigned Halving Add. This instruction adds corresponding unsigned integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are truncated. For rounded results, see *URHADD*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = (element1 + element2) >> 1;
    Elem[result, e, esize] = sum<esize-1:0>;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

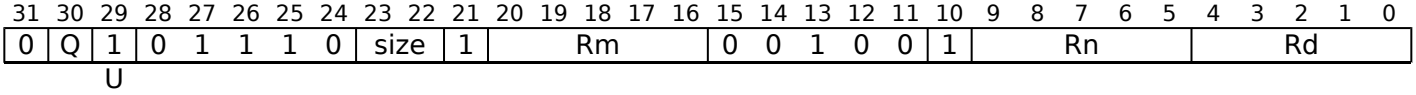
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UHSUB

Unsigned Halving Subtract. This instruction subtracts the vector elements in the second source SIMD&FP register from the corresponding vector elements in the first source SIMD&FP register, shifts each result right one bit, places each result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UHSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer diff;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = (element1 - element2) >> 1;
    Elem[result, e, esize] = diff<esize-1:0>;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

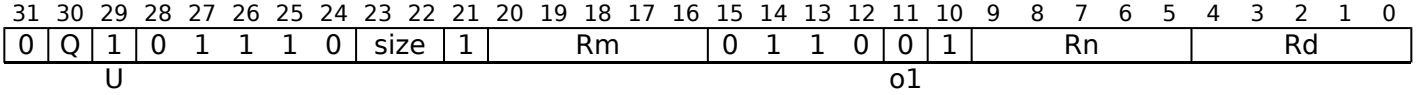
- The execution time of this instruction is independent of:

    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

UMAX

Unsigned Maximum (vector). This instruction compares corresponding elements in the vectors in the two source SIMD&FP registers, places the larger of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UMAX <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

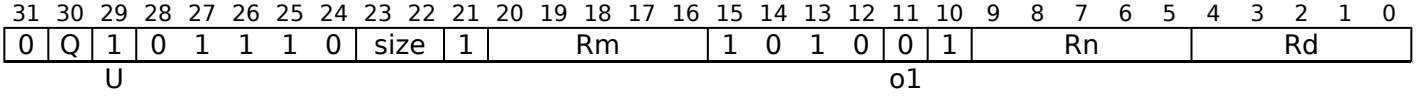
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMAXP

Unsigned Maximum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the largest of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UMAXP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[concat, 2*e, esize], unsigned);
    element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

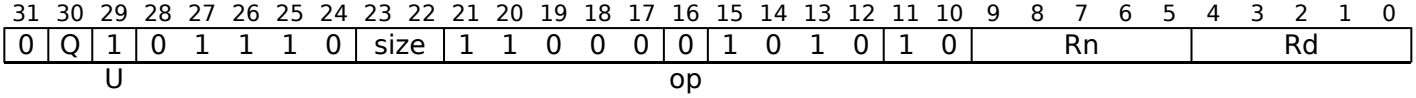
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMAXV

Unsigned Maximum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the largest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UMAXV <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');
```

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
integer maxmin;
integer element;

maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);

V[d, esize] = maxmin<esize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

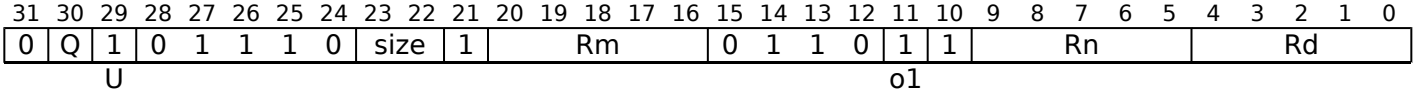
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMIN

Unsigned Minimum (vector). This instruction compares corresponding vector elements in the two source SIMD&FP registers, places the smaller of each of the two unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UMIN <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

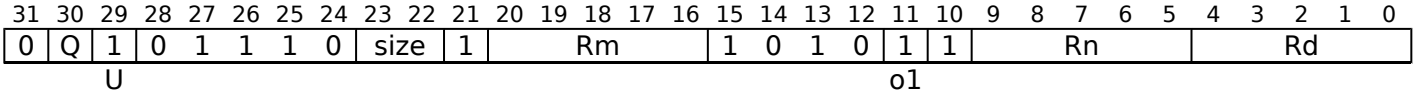
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMINP

Unsigned Minimum Pairwise. This instruction creates a vector by concatenating the vector elements of the first source SIMD&FP register after the vector elements of the second source SIMD&FP register, reads each pair of adjacent vector elements in the two source SIMD&FP registers, writes the smallest of each pair of unsigned integer values into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UMINP <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean minimum = (o1 == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
bits(2*datasize) concat = operand2:operand1;
integer element1;
integer element2;
integer maxmin;

for e = 0 to elements-1
    element1 = Int(Elem[concat, 2*e, esize], unsigned);
    element2 = Int(Elem[concat, (2*e)+1, esize], unsigned);
    maxmin = if minimum then Min(element1, element2) else Max(element1, element2);
    Elem[result, e, esize] = maxmin<esize-1:0>;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

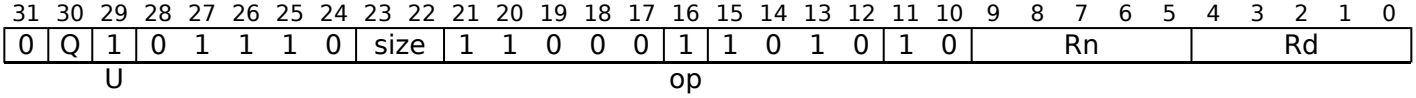
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMINV

Unsigned Minimum across Vector. This instruction compares all the vector elements in the source SIMD&FP register, and writes the smallest of the values as a scalar to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UMINV <V><d>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '100' then UNDEFINED;
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean min = (op == '1');
```

Assembler Symbols

<V> Is the destination width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	RESERVED
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
integer maxmin;
integer element;

maxmin = Int(Elem[operand, 0, esize], unsigned);
for e = 1 to elements-1
    element = Int(Elem[operand, e, esize], unsigned);
    maxmin = if min then Min(maxmin, element) else Max(maxmin, element);

V[d, esize] = maxmin<esize-1:0>;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

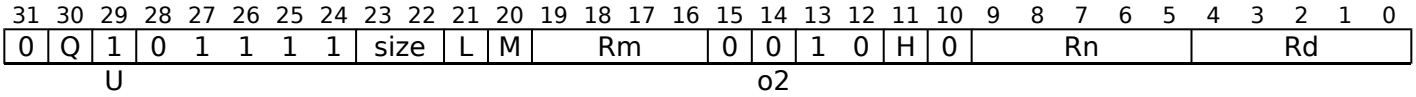
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMLAL, UMLAL2 (by element)

Unsigned Multiply-Add Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The UMLAL instruction extracts vector elements from the lower half of the first source register. The UMLAL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d, 2*datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

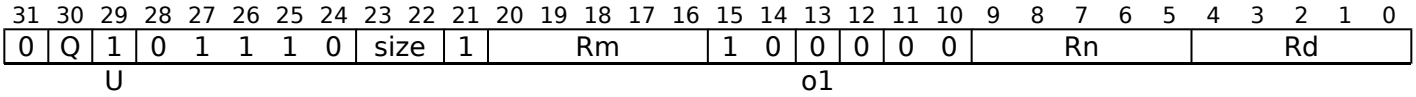
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

UMLAL, UMLAL2 (vector)

Unsigned Multiply-Add Long (vector). This instruction multiplies the vector elements in the lower or upper half of the first source SIMD&FP register by the corresponding vector elements of the second source SIMD&FP register, and accumulates the results with the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The UMLAL instruction extracts vector elements from the lower half of the first source register. The UMLAL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UMLAL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d, 2*datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

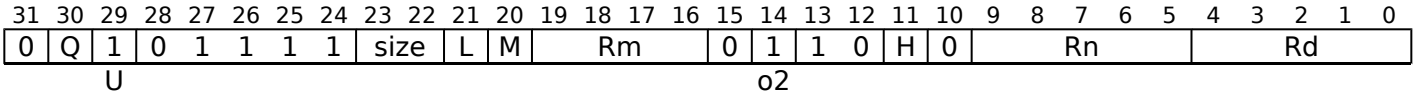
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMLSL, UMLSL2 (by element)

Unsigned Multiply-Subtract Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The `UMLSL` instruction extracts vector elements from the lower half of the first source register. The `UMLSL2` instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



`UMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]`

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
boolean sub_op = (o2 == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] - product;
    else
        Elem[result, e, 2*esize] = Elem[operand3, e, 2*esize] + product;

V[d, 2*datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

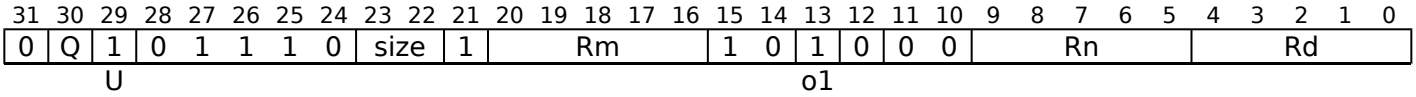
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

UMLSL, UMLSL2 (vector)

Unsigned Multiply-Subtract Long (vector). This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, and subtracts the results from the vector elements of the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are unsigned integer values.

The UMLSL instruction extracts each source vector from the lower half of each source register. The UMLSL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UMLSL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) operand3 = V[d, 2*datasize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;
bits(2*esize) accum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    if sub_op then
        accum = Elem[operand3, e, 2*esize] - product;
    else
        accum = Elem[operand3, e, 2*esize] + product;
    Elem[result, e, 2*esize] = accum;

V[d, 2*datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

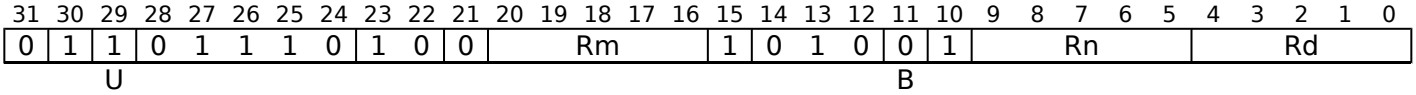
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMMLA (vector)

Unsigned 8-bit integer matrix multiply-accumulate. This instruction multiplies the 2x8 matrix of unsigned 8-bit integer values in the first source vector by the 8x2 matrix of unsigned 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2 to Armv8.5, this is an OPTIONAL instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. *ID_AA64ISAR1_EL1*.I8MM indicates whether this instruction is supported.

Vector
(FEAT_I8MM)



UMMLA <Vd>.4S, <Vn>.16B, <Vm>.16B

```
if !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(128) operand1 = V[n, 128];
bits(128) operand2 = V[m, 128];
bits(128) addend = V[d, 128];

V[d, 128] = MatMulAdd(addend, operand1, operand2, TRUE, TRUE);
```

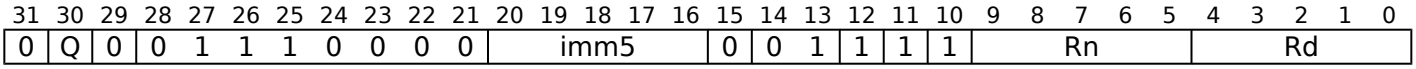
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMOV

Unsigned Move vector element to general-purpose register. This instruction reads the unsigned integer from the source SIMD&FP register, zero-extends it to form a 32-bit or 64-bit value, and writes the result to the destination general-purpose register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias MOV (to general).



32-bit (Q == 0)

```
UMOV <Wd>, <Vn>.<Ts>[<index>]
```

64-bit (Q == 1 && imm5 == x1000)

```
UMOV <Xd>, <Vn>.<Ts>[<index>]
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size;
case Q:imm5 of
    when '0xxxx1' size = 0;    // UMOV Wd, Vn.B
    when '0xxx10' size = 1;    // UMOV Wd, Vn.H
    when '0xx100' size = 2;    // UMOV Wd, Vn.S
    when '1x1000' size = 3;    // UMOV Xd, Vn.D
    otherwise UNDEFINED;

integer idxdsize = if imm5<4> == '1' then 128 else 64;
integer index = UInt(imm5<4:size+1>);
integer esize = 8 << size;
integer datasize = if Q == '1' then 64 else 32;
```

Assembler Symbols

<Wd> Is the 32-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd" field.

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ts> For the 32-bit variant: is an element size specifier, encoded in “imm5”:

imm5	<Ts>
xx000	RESERVED
xxxx1	B
xxx10	H
xx100	S

For the 64-bit variant: is an element size specifier, encoded in “imm5”:

imm5	<Ts>
x0000	RESERVED
xxxx1	RESERVED
xxx10	RESERVED
xx100	RESERVED
x1000	D

<index> For the 32-bit variant: is the element index encoded in “imm5”:

imm5	<index>
xx000	RESERVED
xxxx1	imm5<4:1>
xxx10	imm5<4:2>
xx100	imm5<4:3>

For the 64-bit variant: is the element index encoded in "imm5<4>".

Alias Conditions

Alias	Is preferred when
MOV (to general)	`imm5 == 'x1000'`
MOV (to general)	`imm5 == 'xx100'`

Operation

```
if index == 0 then
    CheckFPEnabled64();
else
    CheckFPAdvSIMDEnabled64();
bits(idxdsize) operand = V[n, idxdsize];

X[d, datasize] = ZeroExtend(Elem[operand, index, esize], datasize);
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

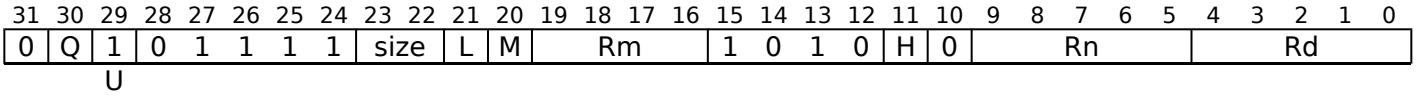
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMULL, UMULL2 (by element)

Unsigned Multiply Long (vector, by element). This instruction multiplies each vector element in the lower or upper half of the first source SIMD&FP register by the specified vector element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied.

The UMULL instruction extracts vector elements from the lower half of the first source register. The UMULL2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Ts>[<index>]

```
integer idxdsize = if H == '1' then 128 else 64;
integer index;
bit Rmhi;
case size of
    when '01' index = UInt(H:L:M); Rmhi = '0';
    when '10' index = UInt(H:L); Rmhi = M;
    otherwise UNDEFINED;

integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rmhi:Rm);

integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	RESERVED
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in “size:M:Rm”:

size	<Vm>
00	RESERVED
01	0:Rm
10	M:Rm
11	RESERVED

Restricted to V0-V15 when element size <Ts> is H.

<Ts> Is an element size specifier, encoded in “size”:

size	<Ts>
00	RESERVED
01	H
10	S
11	RESERVED

<index> Is the element index, encoded in “size:L:H:M”:

size	<index>
00	RESERVED
01	H:L:M
10	H:L
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(idxdsize) operand2 = V[m, idxdsize];
bits(2*datasize) result;
integer element1;
integer element2;
bits(2*esize) product;

element2 = Int(Elem[operand2, index, esize], unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    product = (element1*element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = product;

V[d, 2*datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

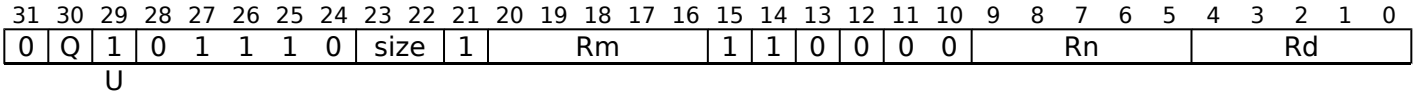
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

UMULL, UMULL2 (vector)

Unsigned Multiply long (vector). This instruction multiplies corresponding vector elements in the lower or upper half of the two source SIMD&FP registers, places the result in a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the elements that are multiplied. All the values in this instruction are unsigned integer values.

The UMULL instruction extracts each source vector from the lower half of each source register. The UMULL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UMULL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    Elem[result, e, 2*esize] = (element1*element2)<2*esize-1:0>;

V[d, 2*datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQADD

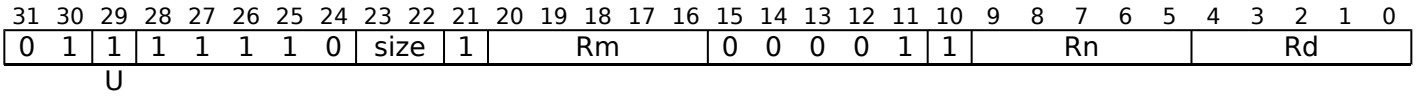
Unsigned saturating Add. This instruction adds the values of corresponding elements of the two source SIMD&FP registers, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

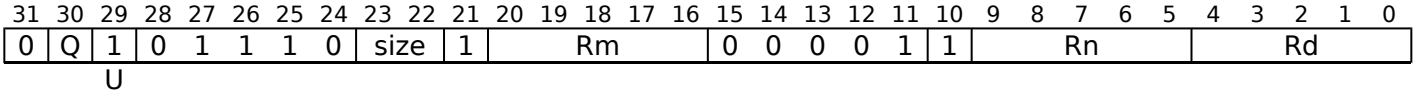
Scalar



UQADD <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Vector



UQADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer sum;
boolean sat;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = element1 + element2;
    (Elem[result, e, esize], sat) = SatQ(sum, esize, unsigned);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQRSHL

Unsigned saturating Rounding Shift Left (register). This instruction takes each vector element of the first source SIMD&FP register, shifts the vector element by a value from the least significant byte of the corresponding vector element of the second source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

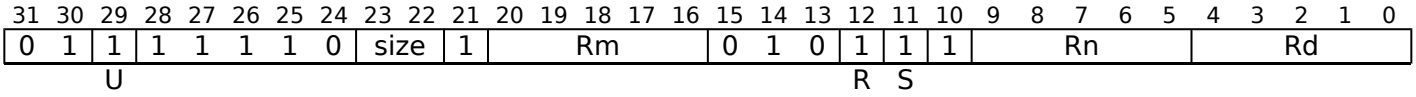
If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are rounded. For truncated results, see *UQSHL*.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

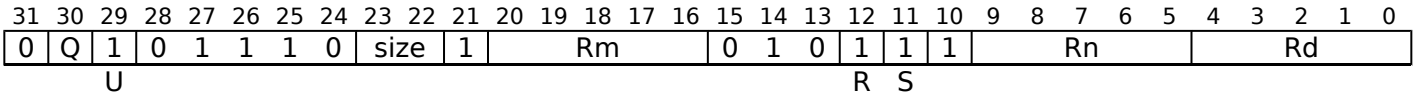
Scalar



UQRSHL <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;
```

Vector



UQRSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

boolean sat;

for e = 0 to elements-1
    integer element = Int(Elem[operand1, e, esize], unsigned);
    integer shift = SInt(Elem[operand2, e, esize]<7:0>);
    if shift >= 0 then // left shift
        element = element << shift;
    else // right shift
        shift = -shift;
        if rounding then
            element = (element + (1 << (shift - 1))) >> shift;
        else
            element = element >> shift;

    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;
```

UQRSHRN, UQRSHRN2

Unsigned saturating Rounded Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see *UQSHRN*.

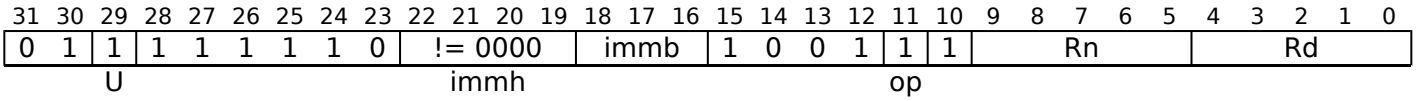
The UQRSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the UQRSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

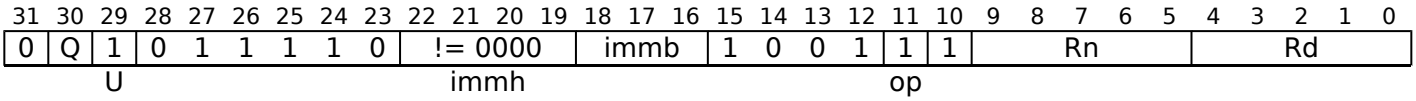
Scalar



UQRSHRN <Vb><d>, <Va><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

Vector



UQRSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vb> Is the destination width specifier, encoded in “immh”:

immh	<Vb>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in “immh”:

immh	<Va>
0000	RESERVED
0001	H
001x	S
01xx	D
1xxx	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

immh	<shift>
0000	RESERVED
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n, datasize*2];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
    element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQSHL (immediate)

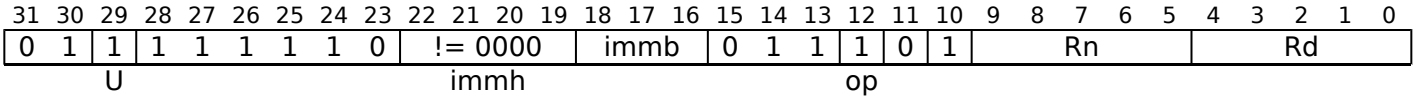
Unsigned saturating Shift Left (immediate). This instruction takes each vector element in the source SIMD&FP register, shifts it by an immediate value, places the results in a vector, and writes the vector to the destination SIMD&FP register. The results are truncated. For rounded results, see *UQRSHL*.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

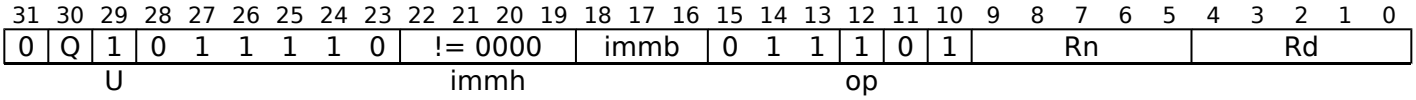
Scalar



UQSHL <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
    when '00' UNDEFINED;
    when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
    when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
    when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```

Vector



UQSHL <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;

boolean src_unsigned;
boolean dst_unsigned;
case op:U of
    when '00' UNDEFINED;
    when '01' src_unsigned = FALSE; dst_unsigned = TRUE;
    when '10' src_unsigned = FALSE; dst_unsigned = FALSE;
    when '11' src_unsigned = TRUE; dst_unsigned = TRUE;
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the left shift amount, in the range 0 to the operand width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	RESERVED
0001	(UInt(immh:immb)-8)
001x	(UInt(immh:immb)-16)
01xx	(UInt(immh:immb)-32)
1xxx	(UInt(immh:immb)-64)

For the vector variant: is the left shift amount, in the range 0 to the element width in bits minus 1, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(UInt(immh:immb)-8)
001x	(UInt(immh:immb)-16)
01xx	(UInt(immh:immb)-32)
1xxx	(UInt(immh:immb)-64)

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
integer element;
boolean sat;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], src_unsigned) << shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, dst_unsigned);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```

UQSHL (register)

Unsigned saturating Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts the element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

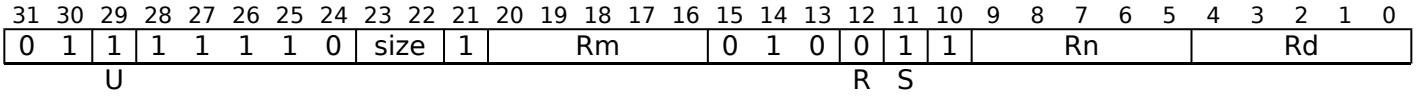
If the shift value is positive, the operation is a left shift. Otherwise, it is a right shift. The results are truncated. For rounded results, see *UQRSHL*.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

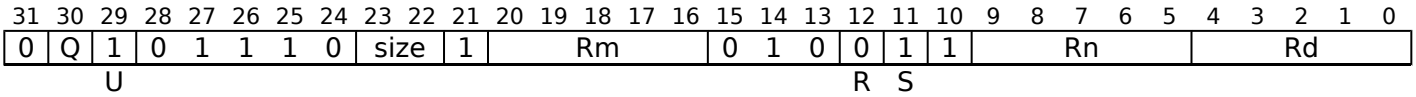
Scalar



UQSHL <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;
```

Vector



UQSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

boolean sat;

for e = 0 to elements-1
    integer element = Int(Elem[operand1, e, esize], unsigned);
    integer shift = SInt(Elem[operand2, e, esize]<7:0>);
    if shift >= 0 then // left shift
        element = element << shift;
    else // right shift
        shift = -shift;
        if rounding then
            element = (element + (1 << (shift - 1))) >> shift;
        else
            element = element >> shift;

    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;
```

UQSHRN, UQSHRN2

Unsigned saturating Shift Right Narrow (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, saturates each shifted result to a value that is half the original width, puts the final result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see *UQRSHRN*.

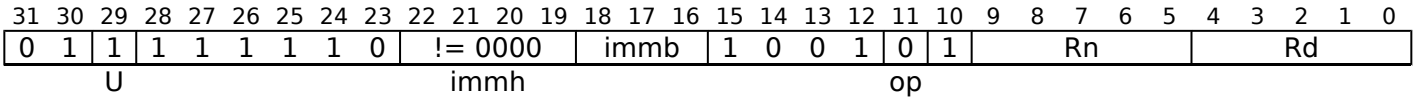
The UQSHRN instruction writes the vector to the lower half of the destination register and clears the upper half, while the UQSHRN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

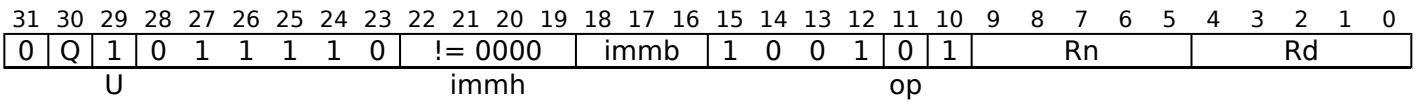
Scalar



UQSHRN <Vb><d>, <Va><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then UNDEFINED;
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = esize;
integer elements = 1;
integer part = 0;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

Vector



UQSHRN{2} <Vd>.<Tb>, <Vn>.<Ta>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = (2 * esize) - UInt(immh:immb);
boolean round = (op == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vb> Is the destination width specifier, encoded in “immh”:

immh	<Vb>
0000	RESERVED
0001	B
001x	H
01xx	S
1xxx	RESERVED

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<Va> Is the source width specifier, encoded in “immh”:

immh	<Va>
0000	RESERVED
0001	H
001x	S
01xx	D
1xxx	RESERVED

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to the destination operand width in bits, encoded in “immh:immb”:

immh	<shift>
0000	RESERVED
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

For the vector variant: is the right shift amount, in the range 1 to the destination element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize*2) operand = V[n, datasize*2];
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;
boolean sat;

for e = 0 to elements-1
    element = (Int(Elem[operand, e, 2*esize], unsigned) + round_const) >> shift;
    (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQSUB

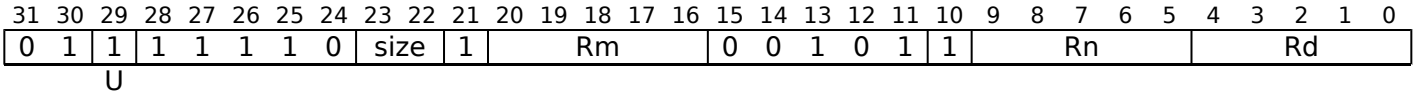
Unsigned saturating Subtract. This instruction subtracts the element values of the second source SIMD&FP register from the corresponding element values of the first source SIMD&FP register, places the results into a vector, and writes the vector to the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

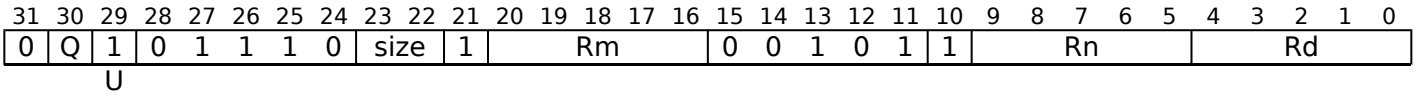
Scalar



UQSUB <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
```

Vector



UQSUB <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer diff;
boolean sat;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    diff = element1 - element2;
    (Elem[result, e, esize], sat) = SatQ(diff, esize, unsigned);
    if sat then FPSR.QC = '1';

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQXTN, UQXTN2

Unsigned saturating extract Narrow. This instruction reads each vector element from the source SIMD&FP register, saturates each value to half the original width, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

The UQXTN instruction writes the vector to the lower half of the destination register and clears the upper half, while the UQXTN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

Scalar

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	1	0	size		1	0	0	0	0	1	0	1	0	0	1	0	Rn					Rd				
		U																													

UQXTN <Vb><d>, <Va><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = esize;
integer part = 0;
integer elements = 1;

boolean unsigned = (U == '1');
```

Vector

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	size		1	0	0	0	0	1	0	1	0	0	1	0	Rn					Rd				
		U																													

UQXTN{2} <Vd>.<Tb>, <Vn>.<Ta>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vb> Is the destination width specifier, encoded in “size”:

size	<Vb>
00	B
01	H
10	S
11	RESERVED

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<Va> Is the source width specifier, encoded in “size”:

size	<Va>
00	H
01	S
10	D
11	RESERVED

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

Operation

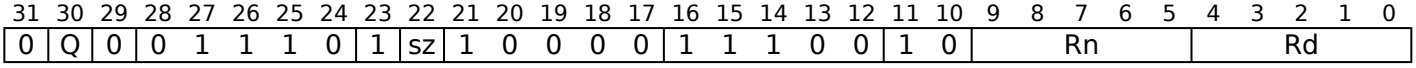
```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n, 2*datasize];
bits(datasize) result;
bits(2*esize) element;
boolean sat;

for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    (Elem[result, e, esize], sat) = SatQ(Int(element, unsigned), esize, unsigned);
    if sat then FPSR.QC = '1';

Vpart[d, part, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

URECPE

Unsigned Reciprocal Estimate. This instruction reads each vector element from the source SIMD&FP register, calculates an approximate inverse for the unsigned integer value, places the result into a vector, and writes the vector to the destination SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



URECPE <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(32) element;

for e = 0 to elements-1
    element = Elem[operand, e, 32];
    Elem[result, e, 32] = UnsignedRecipEstimate(element);

V[d, datasize] = result;
```

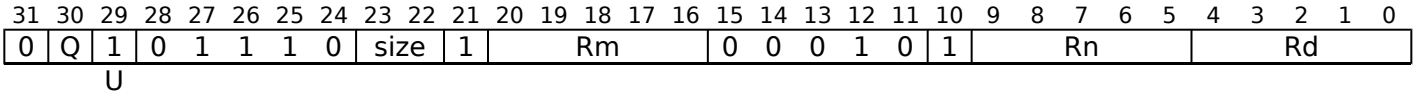
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

URHADD

Unsigned Rounding Halving Add. This instruction adds corresponding unsigned integer values from the two source SIMD&FP registers, shifts each result right one bit, places the results into a vector, and writes the vector to the destination SIMD&FP register.

The results are rounded. For truncated results, see *UHADD*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



URHADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

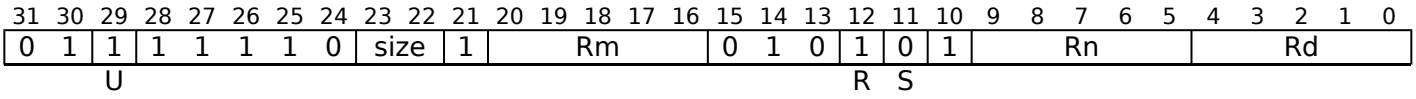
Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    sum = (element1 + element2 + 1) >> 1;
    Elem[result, e, esize] = sum<esize-1:0>;

V[d, datasize] = result;
```

URSHL

Unsigned Rounding Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts the vector element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.
If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift.
Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.
It has encodings from 2 classes: Scalar and Vector

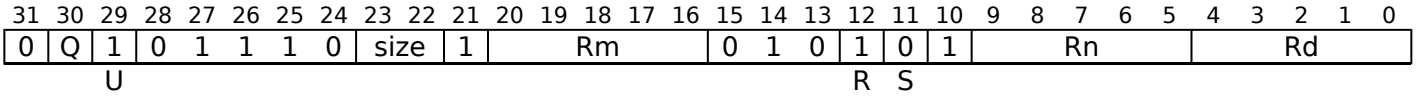
Scalar



URSHL <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;
```

Vector



URSHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

boolean sat;

for e = 0 to elements-1
    integer element = Int(Elem[operand1, e, esize], unsigned);
    integer shift = SInt(Elem[operand2, e, esize]<7:0>);
    if shift >= 0 then // left shift
        element = element << shift;
    else // right shift
        shift = -shift;
        if rounding then
            element = (element + (1 << (shift - 1))) >> shift;
        else
            element = element >> shift;

    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;
```

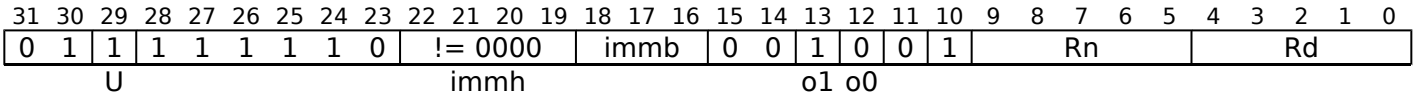
URSHR

Unsigned Rounding Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see *USHR*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

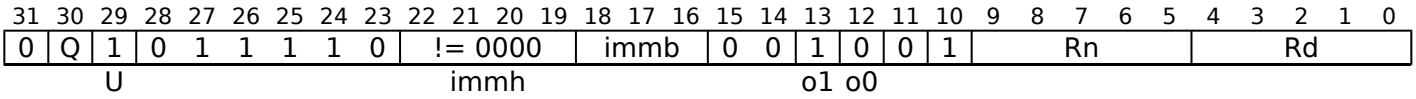
Scalar



URSHR <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector



URSHR <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

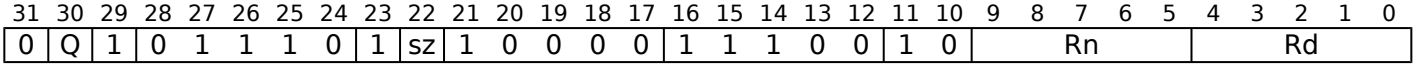
```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

URSQRTE

Unsigned Reciprocal Square Root Estimate. This instruction reads each vector element from the source SIMD&FP register, calculates an approximate inverse square root for each value, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



URSQRTE <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if sz == '1' then UNDEFINED;
integer esize = 32;
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “sz:Q”:

sz	Q	<T>
0	0	2S
0	1	4S
1	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;
bits(32) element;

for e = 0 to elements-1
    element = Elem[operand, e, 32];
    Elem[result, e, 32] = UnsignedRSqrtEstimate(element);

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

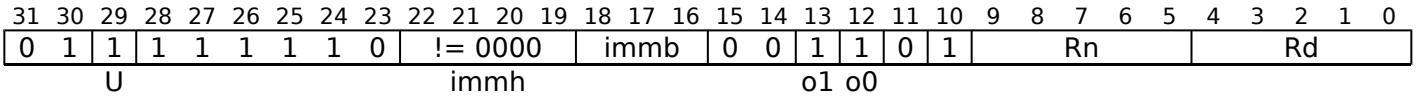
URSRA

Unsigned Rounding Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are rounded. For truncated results, see *USRA*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

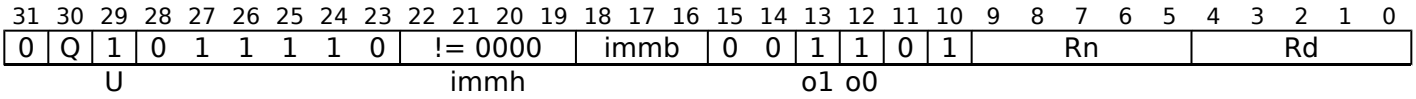
Scalar



URSRA <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector



URSRA <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

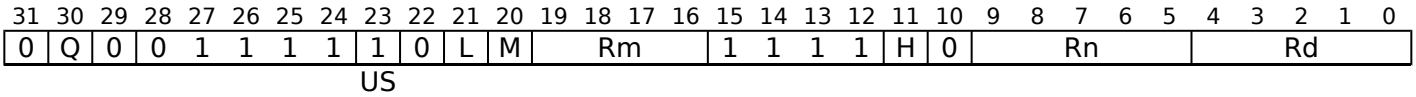
USDOT (by element)

Dot Product index form with unsigned and signed integers. This instruction performs the dot product of the four unsigned 8-bit integer values in each 32-bit element of the first source register with the four signed 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2 to Armv8.5, this is an OPTIONAL instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. *ID_AA64ISAR1_EL1*.I8MM indicates whether this instruction is supported.

Vector

(FEAT_I8MM)



USDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.4B[<index>]

```
if !HaveInt8MatMulExt() then UNDEFINED;
boolean op1_unsigned = (US == '1');
boolean op2_unsigned = (US == '0');
integer n = UInt(Rn);
integer m = UInt(M:Rm);
integer d = UInt(Rd);
integer i = UInt(H:L);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	8B
1	16B

<Vm> Is the name of the second SIMD&FP source register, encoded in the "M:Rm" fields.

<index> Is the immediate index of a 32-bit group of four 8-bit values in the range 0 to 3, encoded in the "H:L" fields.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(128) operand2 = V[m, 128];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;

for e = 0 to elements-1
    bits(32) res = Elem[operand3, e, 32];
    for b = 0 to 3
        integer element1 = Int(Elem[operand1, 4*e+b, 8], op1_unsigned);
        integer element2 = Int(Elem[operand2, 4*i+b, 8], op2_unsigned);
        res = res + element1 * element2;
    Elem[result, e, 32] = res;
V[d, datasize] = result;
```

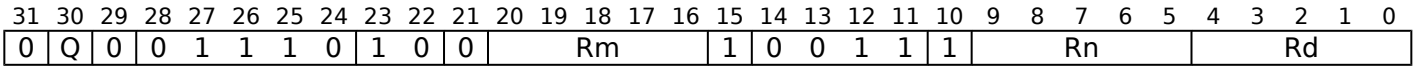
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

USDOT (vector)

Dot Product vector form with unsigned and signed integers. This instruction performs the dot product of the four unsigned 8-bit integer values in each 32-bit element of the first source register with the four signed 8-bit integer values in the corresponding 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2 to Armv8.5, this is an OPTIONAL instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. *ID_AA64ISAR1_EL1*.I8MM indicates whether this instruction is supported.

Vector
(FEAT_I8MM)



USDOT <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
if !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV 32;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	8B
1	16B

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;

for e = 0 to elements-1
    bits(32) res = Elem[operand3, e, 32];
    for b = 0 to 3
        integer element1 = UInt(Elem[operand1, 4*e+b, 8]);
        integer element2 = SInt(Elem[operand2, 4*e+b, 8]);
        res = res + element1 * element2;
    Elem[result, e, 32] = res;

V[d, datasize] = result;
```

USHL

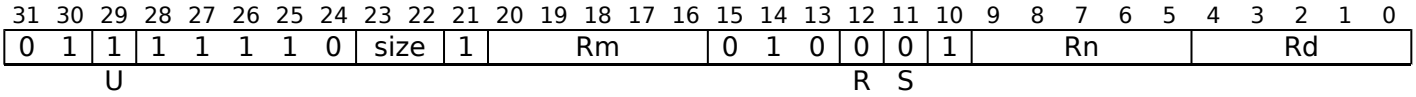
Unsigned Shift Left (register). This instruction takes each element in the vector of the first source SIMD&FP register, shifts each element by a value from the least significant byte of the corresponding element of the second source SIMD&FP register, places the results in a vector, and writes the vector to the destination SIMD&FP register.

If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a truncating right shift. For a rounding shift, see *URSHL*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

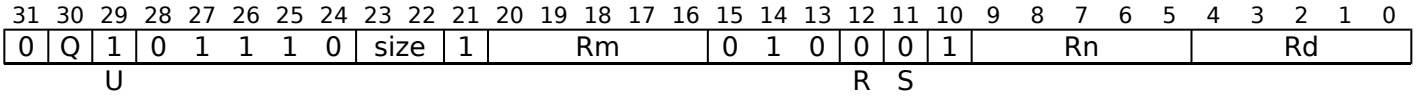
Scalar



USHL <V><d>, <V><n>, <V><m>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
if S == '0' && size != '11' then UNDEFINED;
```

Vector



USHL <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
boolean unsigned = (U == '1');
boolean rounding = (R == '1');
boolean saturating = (S == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
0x	RESERVED
10	RESERVED
11	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<m> Is the number of the second SIMD&FP source register, encoded in the "Rm" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

boolean sat;

for e = 0 to elements-1
    integer element = Int(Elem[operand1, e, esize], unsigned);
    integer shift = SInt(Elem[operand2, e, esize]<7:0>);
    if shift >= 0 then // left shift
        element = element << shift;
    else // right shift
        shift = -shift;
        if rounding then
            element = (element + (1 << (shift - 1))) >> shift;
        else
            element = element >> shift;

    if saturating then
        (Elem[result, e, esize], sat) = SatQ(element, esize, unsigned);
        if sat then FPSR.QC = '1';
    else
        Elem[result, e, esize] = element<esize-1:0>;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

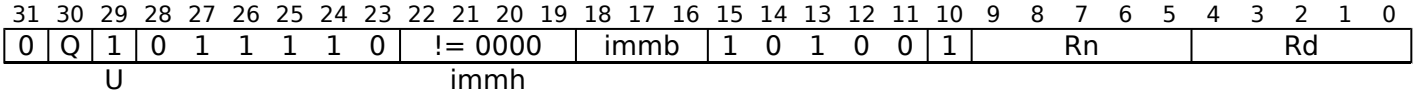
USHLL, USHLL2

Unsigned Shift Left Long (immediate). This instruction reads each vector element in the lower or upper half of the source SIMD&FP register, shifts the unsigned integer value left by the specified number of bits, places the result into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The USHLL instruction extracts vector elements from the lower half of the source register. The USHLL2 instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This instruction is used by the alias UXTL, UXTL2.



USHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3> == '1' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

integer shift = UInt(immh:immb) - esize;
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

<shift> Is the left shift amount, in the range 0 to the source element width in bits minus 1, encoded in "immh:immb":

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(UInt(immh:immb)-8)
001x	(UInt(immh:immb)-16)
01xx	(UInt(immh:immb)-32)
1xxx	RESERVED

Alias Conditions

Alias	Is preferred when
UXTL, UXTL2	immb == '000' && BitCount(immh) == 1

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = Vpart[n, part, datasize];
bits(datasize*2) result;
integer element;

for e = 0 to elements-1
    element = Int(Elem[operand, e, esize], unsigned) << shift;
    Elem[result, e, 2*esize] = element<2*esize-1:0>;

V[d, datasize*2] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

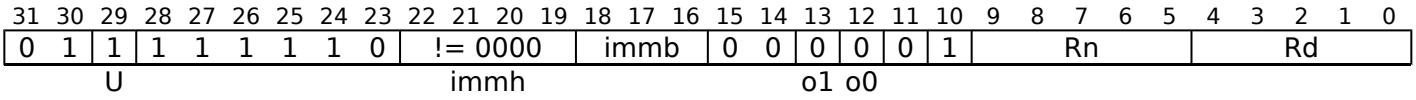
USHR

Unsigned Shift Right (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see *URSHR*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

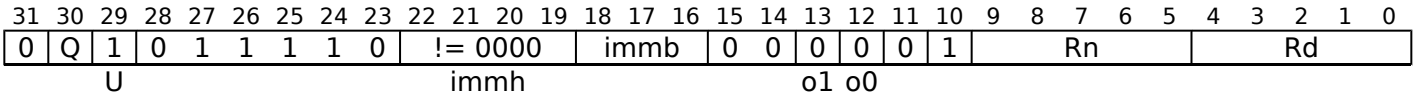
Scalar



USHR <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector



USHR <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

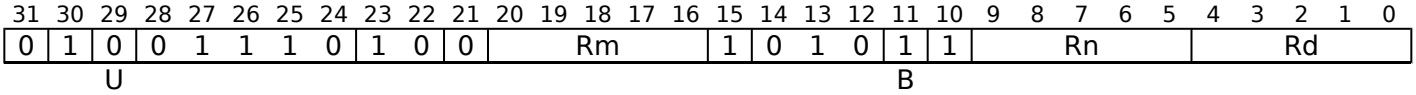
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

USMMLA (vector)

Unsigned and signed 8-bit integer matrix multiply-accumulate. This instruction multiplies the 2x8 matrix of unsigned 8-bit integer values in the first source vector by the 8x2 matrix of signed 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2 to Armv8.5, this is an OPTIONAL instruction. From Armv8.6 it is mandatory for implementations that include Advanced SIMD to support it. *ID_AA64ISAR1_EL1*.I8MM indicates whether this instruction is supported.

Vector
(FEAT_I8MM)



USMMLA <Vd>.4S, <Vn>.16B, <Vm>.16B

```
if !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Rd);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP third source and destination register, encoded in the "Rd" field.
<Vn>	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(128) operand1 = V[n, 128];
bits(128) operand2 = V[m, 128];
bits(128) addend = V[d, 128];

V[d, 128] = MatMulAdd(addend, operand1, operand2, TRUE, FALSE);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

USQADD

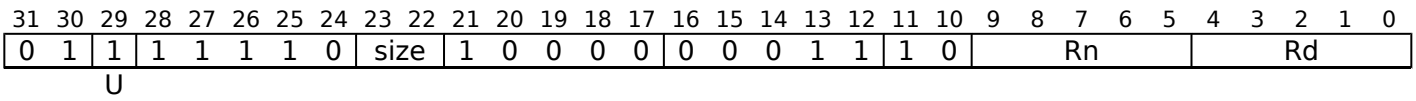
Unsigned saturating Accumulate of Signed value. This instruction adds the signed integer values of the vector elements in the source SIMD&FP register to corresponding unsigned integer values of the vector elements in the destination SIMD&FP register, and accumulates the resulting unsigned integer values with the vector elements of the destination SIMD&FP register.

If overflow occurs with any of the results, those results are saturated. If saturation occurs, the cumulative saturation bit *FPSR*.QC is set.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

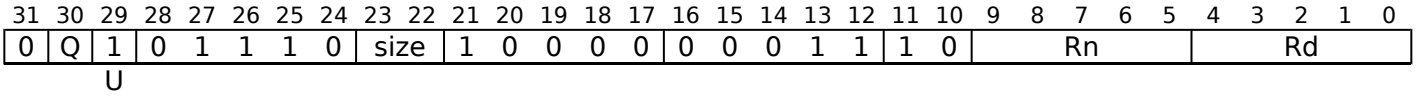
Scalar



USQADD <V><d>, <V><n>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer esize = 8 << UInt(size);
integer datasize = esize;
integer elements = 1;

boolean unsigned = (U == '1');
```

Vector



USQADD <Vd>.<T>, <Vn>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

boolean unsigned = (U == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number of the SIMD&FP destination register, encoded in the "Rd" field.

<n> Is the number of the SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) result;

bits(datasize) operand2 = V[d, datasize];
integer op1;
integer op2;
boolean sat;

for e = 0 to elements-1
    op1 = Int(Elem[operand, e, esize], !unsigned);
    op2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], sat) = SatQ(op1 + op2, esize, unsigned);
    if sat then FPSR.QC = '1';
V[d, datasize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

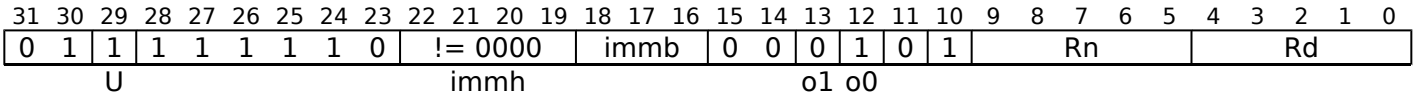
USRA

Unsigned Shift Right and Accumulate (immediate). This instruction reads each vector element in the source SIMD&FP register, right shifts each result by an immediate value, and accumulates the final results with the vector elements of the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The results are truncated. For rounded results, see *URSRA*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: Scalar and Vector

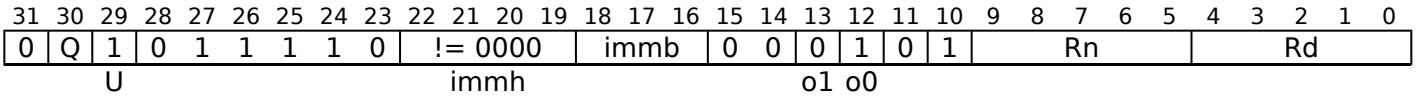
Scalar



USRA <V><d>, <V><n>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh<3> != '1' then UNDEFINED;
integer esize = 8 << 3;
integer datasize = esize;
integer elements = 1;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Vector



USRA <Vd>.<T>, <Vn>.<T>, #<shift>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if immh == '0000' then SEE(asimdimm);
if immh<3>:Q == '10' then UNDEFINED;
integer esize = 8 << HighestSetBit(immh);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;

integer shift = (esize * 2) - UInt(immh:immb);
boolean unsigned = (U == '1');
boolean round = (o1 == '1');
boolean accumulate = (o0 == '1');
```

Assembler Symbols

<V> Is a width specifier, encoded in “immh”:

immh	<V>
0xxx	RESERVED
1xxx	D

<d> Is the number of the SIMD&FP destination register, in the "Rd" field.

<n> Is the number of the first SIMD&FP source register, encoded in the "Rn" field.

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<T>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	0	RESERVED
1xxx	1	2D

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<shift> For the scalar variant: is the right shift amount, in the range 1 to 64, encoded in “immh:immb”:

immh	<shift>
0xxx	RESERVED
1xxx	(128-UInt(immh:immb))

For the vector variant: is the right shift amount, in the range 1 to the element width in bits, encoded in “immh:immb”:

immh	<shift>
0000	SEE Advanced SIMD modified immediate
0001	(16-UInt(immh:immb))
001x	(32-UInt(immh:immb))
01xx	(64-UInt(immh:immb))
1xxx	(128-UInt(immh:immb))

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand = V[n, datasize];
bits(datasize) operand2;
bits(datasize) result;
integer round_const = if round then (1 << (shift - 1)) else 0;
integer element;

operand2 = if accumulate then V[d, datasize] else Zeros(datasize);
for e = 0 to elements-1
    element = (Int(Elem[operand, e, esize], unsigned) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

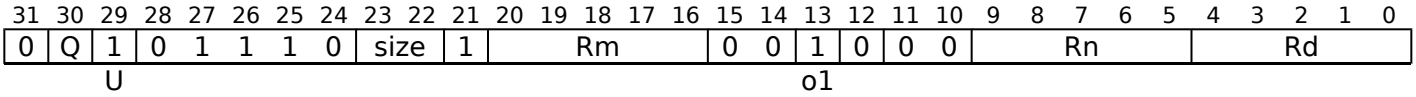
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

USUBL, USUBL2

Unsigned Subtract Long. This instruction subtracts each vector element in the lower or upper half of the second source SIMD&FP register from the corresponding vector element of the first source SIMD&FP register, places the result into a vector, and writes the vector to the destination SIMD&FP register. All the values in this instruction are unsigned integer values. The destination vector elements are twice as long as the source vector elements.

The USUBL instruction extracts each source vector from the lower half of each source register. The USUBL2 instruction extracts each source vector from the upper half of each source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



USUBL{2} <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = Vpart[n, part, datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d, 2*datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

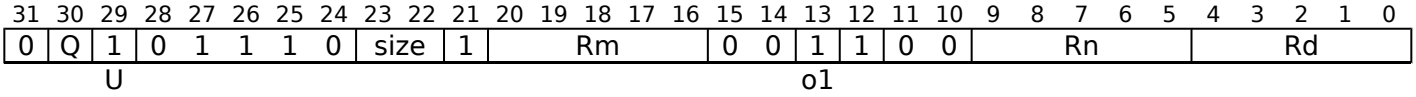
USUBW, USUBW2

Unsigned Subtract Wide. This instruction subtracts each vector element of the second source SIMD&FP register from the corresponding vector element in the lower or upper half of the first source SIMD&FP register, places the result in a vector, and writes the vector to the SIMD&FP destination register. All the values in this instruction are unsigned integer values.

The vector elements of the destination register and the first source register are twice as long as the vector elements of the second source register.

The USUBW instruction extracts vector elements from the lower half of the first source register. The USUBW2 instruction extracts vector elements from the upper half of the first source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



USUBW{2} <Vd>.<Ta>, <Vn>.<Ta>, <Vm>.<Tb>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (o1 == '1');
boolean unsigned = (U == '1');
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand1 = V[n, 2*datasize];
bits(datasize) operand2 = Vpart[m, part, datasize];
bits(2*datasize) result;
integer element1;
integer element2;
integer sum;

for e = 0 to elements-1
    element1 = Int(Elem[operand1, e, 2*esize], unsigned);
    element2 = Int(Elem[operand2, e, esize], unsigned);
    if sub_op then
        sum = element1 - element2;
    else
        sum = element1 + element2;
    Elem[result, e, 2*esize] = sum<2*esize-1:0>;

V[d, 2*datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UXTL, UXTL2

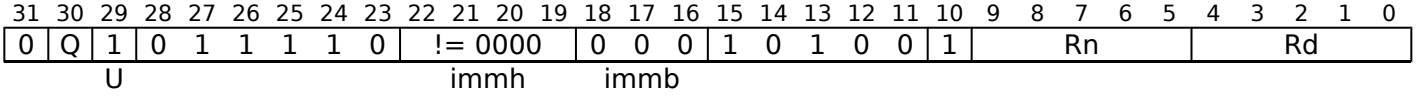
Unsigned extend Long. This instruction copies each vector element from the lower or upper half of the source SIMD&FP register into a vector, and writes the vector to the destination SIMD&FP register. The destination vector elements are twice as long as the source vector elements.

The UXTL instruction extracts vector elements from the lower half of the source register. The UXTL2 instruction extracts vector elements from the upper half of the source register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

This is an alias of USHLL, USHLL2. This means:

- The encodings in this description are named to match the encodings of USHLL, USHLL2.
- The description of USHLL, USHLL2 gives the operational pseudocode for this instruction.



UXTL{2} <Vd>.<Ta>, <Vn>.<Tb>

is equivalent to

USHLL{2} <Vd>.<Ta>, <Vn>.<Tb>, #0

and is the preferred disassembly when `BitCount(immh) == 1`.

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Ta> Is an arrangement specifier, encoded in “immh”:

immh	<Ta>
0000	SEE Advanced SIMD modified immediate
0001	8H
001x	4S
01xx	2D
1xxx	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Tb> Is an arrangement specifier, encoded in “immh:Q”:

immh	Q	<Tb>
0000	x	SEE Advanced SIMD modified immediate
0001	0	8B
0001	1	16B
001x	0	4H
001x	1	8H
01xx	0	2S
01xx	1	4S
1xxx	x	RESERVED

Operation

The description of USHLL, USHLL2 gives the operational pseudocode for this instruction.

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

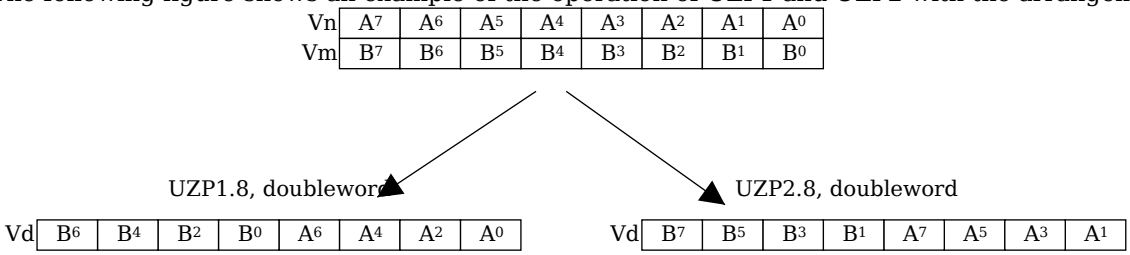
UZP1

Unzip vectors (primary). This instruction reads corresponding even-numbered vector elements from the two source SIMD&FP registers, starting at zero, places the result from the first source register into consecutive elements in the lower half of a vector, and writes the vector to the destination SIMD&FP register.

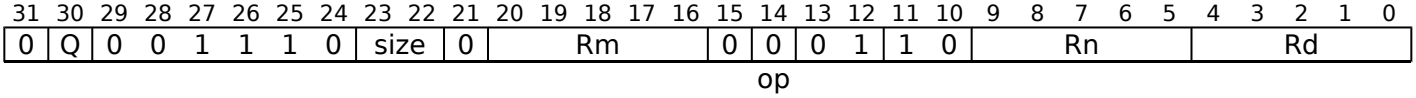
Note

This instruction can be used with UZP2 to de-interleave two vectors.

The following figure shows an example of the operation of UZP1 and UZP2 with the arrangement specifier 8B.



Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UZP1 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operandl = V[n, datasize];
bits(datasize) operandh = V[m, datasize];
bits(datasize) result;

bits(datasize*2) zipped = operandh:operandl;
for e = 0 to elements-1
    Elem[result, e, esize] = Elem[zipped, 2*e+part, esize];

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

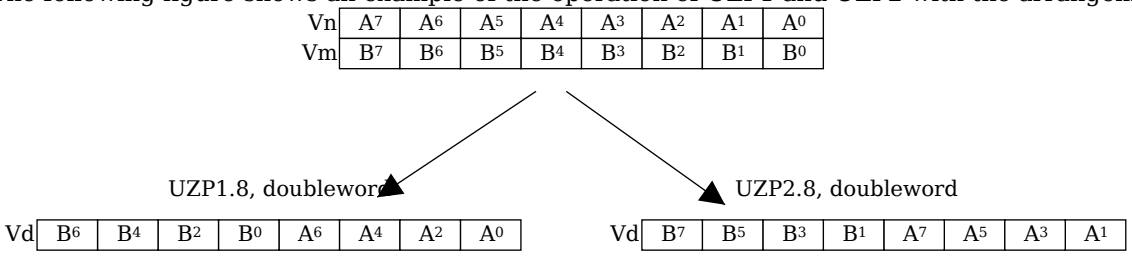
UZP2

Unzip vectors (secondary). This instruction reads corresponding odd-numbered vector elements from the two source SIMD&FP registers, places the result from the first source register into consecutive elements in the lower half of a vector, and writes the vector to the destination SIMD&FP register.

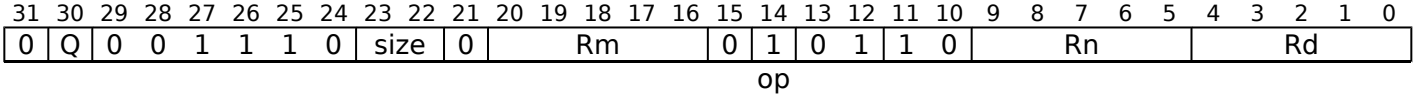
Note

This instruction can be used with UZP1 to de-interleave two vectors.

The following figure shows an example of the operation of UZP1 and UZP2 with the arrangement specifier 8B.



Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



UZP2 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operandl = V[n, datasize];
bits(datasize) operandh = V[m, datasize];
bits(datasize) result;

bits(datasize*2) zipped = operandh:operandl;
for e = 0 to elements-1
    Elem[result, e, esize] = Elem[zipped, 2*e+part, esize];

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

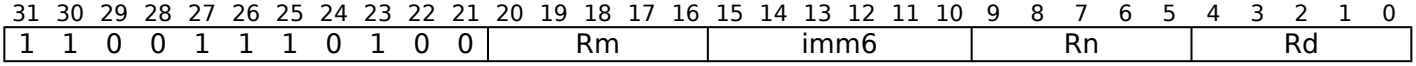
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

XAR

Exclusive OR and Rotate performs a bitwise exclusive OR of the 128-bit vectors in the two source SIMD&FP registers, rotates each 64-bit element of the resulting 128-bit vector right by the value specified by a 6-bit immediate value, and writes the result to the destination SIMD&FP register.

This instruction is implemented only when *FEAT_SHA3* is implemented.

Advanced SIMD
(FEAT_SHA3)



XAR <Vd>.2D, <Vn>.2D, <Vm>.2D, #<imm6>

```
if !HaveSHA3Ext() then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
```

Assembler Symbols

<Vd>	Is the name of the SIMD&FP destination register, encoded in the "Rd" field.
<Vn>	Is the name of the first SIMD&FP source register, encoded in the "Rn" field.
<Vm>	Is the name of the second SIMD&FP source register, encoded in the "Rm" field.
<imm6>	Is a rotation right, encoded in "imm6".

Operation

```
AArch64.CheckFPAdvSIMDEnabled();

bits(128) Vm = V[m, 128];
bits(128) Vn = V[n, 128];
bits(128) tmp;
tmp = Vn EOR Vm;
V[d, 128] = ROR(tmp<127:64>, UInt(imm6)):ROR(tmp<63:0>, UInt(imm6));
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

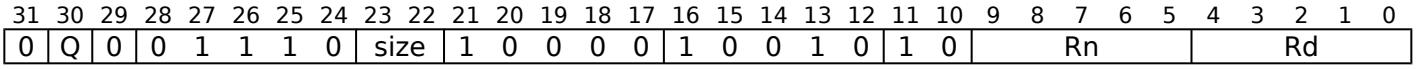
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

XTN, XTN2

Extract Narrow. This instruction reads each vector element from the source SIMD&FP register, narrows each value to half the original width, places the result into a vector, and writes the vector to the lower or upper half of the destination SIMD&FP register. The destination vector elements are half as long as the source vector elements.

The XTN instruction writes the vector to the lower half of the destination register and clears the upper half, while the XTN2 instruction writes the vector to the upper half of the destination register without affecting the other bits of the register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



XTN{2} <Vd>.<Tb>, <Vn>.<Ta>

```
integer d = UInt(Rd);
integer n = UInt(Rn);

if size == '11' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = 64;
integer part = UInt(Q);
integer elements = datasize DIV esize;
```

Assembler Symbols

2 Is the second and upper half specifier. If present it causes the operation to be performed on the upper 64 bits of the registers holding the narrower elements, and is encoded in “Q”:

Q	2
0	[absent]
1	[present]

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<Tb> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<Tb>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	x	RESERVED

<Vn> Is the name of the SIMD&FP source register, encoded in the "Rn" field.

<Ta> Is an arrangement specifier, encoded in “size”:

size	<Ta>
00	8H
01	4S
10	2D
11	RESERVED

Operation

```
CheckFPAdvSIMDEnabled64();
bits(2*datasize) operand = V[n, 2*datasize];
bits(datasize) result;
bits(2*esize) element;

for e = 0 to elements-1
    element = Elem[operand, e, 2*esize];
    Elem[result, e, esize] = element<esize-1:0>;
Vpart[d, part, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

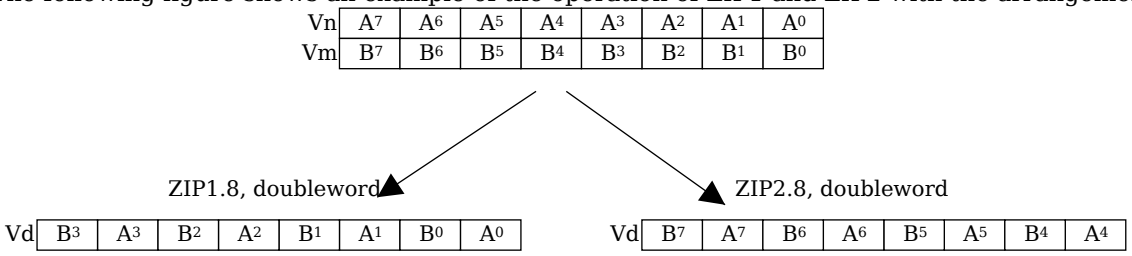
ZIP1

Zip vectors (primary). This instruction reads adjacent vector elements from the lower half of two source SIMD&FP registers as pairs, interleaves the pairs and places them into a vector, and writes the vector to the destination SIMD&FP register. The first pair from the first source register is placed into the two lowest vector elements, with subsequent pairs taken alternately from each source register.

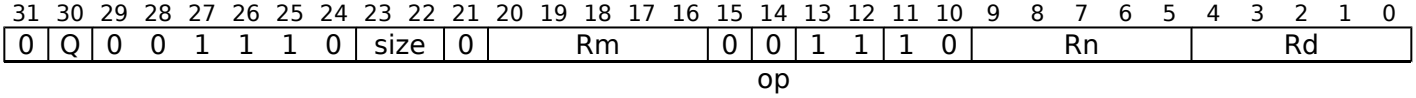
Note

This instruction can be used with ZIP2 to interleave two vectors.

The following figure shows an example of the operation of ZIP1 and ZIP2 with the arrangement specifier 8B.



Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



ZIP1 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

integer base = part * pairs;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, base+p, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, base+p, esize];

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

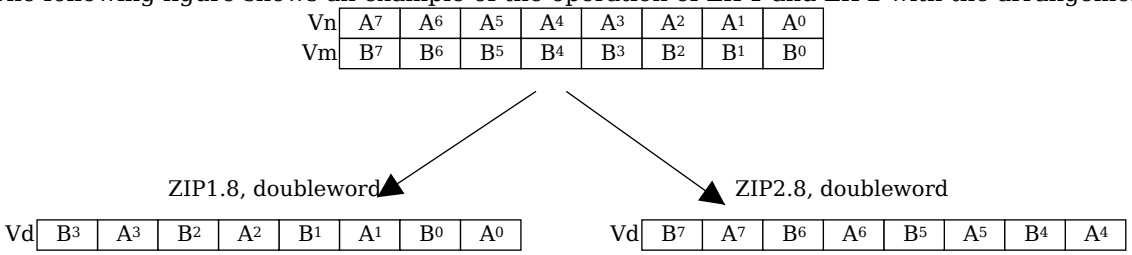
ZIP2

Zip vectors (secondary). This instruction reads adjacent vector elements from the upper half of two source SIMD&FP registers as pairs, interleaves the pairs and places them into a vector, and writes the vector to the destination SIMD&FP register. The first pair from the first source register is placed into the two lowest vector elements, with subsequent pairs taken alternately from each source register.

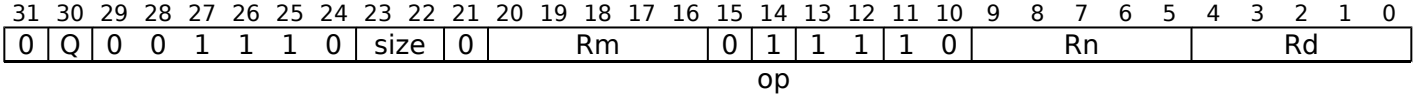
Note

This instruction can be used with ZIP1 to interleave two vectors.

The following figure shows an example of the operation of ZIP1 and ZIP2 with the arrangement specifier 8B.



Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



ZIP2 <Vd>.<T>, <Vn>.<T>, <Vm>.<T>

```
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);

if size:Q == '110' then UNDEFINED;
integer esize = 8 << UInt(size);
integer datasize = if Q == '1' then 128 else 64;
integer elements = datasize DIV esize;
integer part = UInt(op);
integer pairs = elements DIV 2;
```

Assembler Symbols

<Vd> Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T> Is an arrangement specifier, encoded in “size:Q”:

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn> Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm> Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize) operand1 = V[n, datasize];
bits(datasize) operand2 = V[m, datasize];
bits(datasize) result;

integer base = part * pairs;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, base+p, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, base+p, esize];

V[d, datasize] = result;
```

Operational information

If PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

A64 -- SVE Instructions (alphabetic order)

ABS: Absolute value (predicated).

ADCLB: Add with carry long (bottom).

ADCLT: Add with carry long (top).

ADD (immediate): Add immediate (unpredicated).

ADD (vectors, predicated): Add vectors (predicated).

ADD (vectors, unpredicated): Add vectors (unpredicated).

ADDHNB: Add narrow high part (bottom).

ADDHNT: Add narrow high part (top).

ADDP: Add pairwise.

ADDPL: Add multiple of predicate register size to scalar register.

ADDVL: Add multiple of vector register size to scalar register.

ADR: Compute vector address.

AESD: AES single round decryption.

AESE: AES single round encryption.

AESIMC: AES inverse mix columns.

AESMC: AES mix columns.

AND (immediate): Bitwise AND with immediate (unpredicated).

AND (predicates): Bitwise AND predicates.

AND (vectors, predicated): Bitwise AND vectors (predicated).

AND (vectors, unpredicated): Bitwise AND vectors (unpredicated).

ANDS: Bitwise AND predicates, setting the condition flags.

ANDV: Bitwise AND reduction to scalar.

ASR (immediate, predicated): Arithmetic shift right by immediate (predicated).

ASR (immediate, unpredicated): Arithmetic shift right by immediate (unpredicated).

ASR (vectors): Arithmetic shift right by vector (predicated).

ASR (wide elements, predicated): Arithmetic shift right by 64-bit wide elements (predicated).

ASR (wide elements, unpredicated): Arithmetic shift right by 64-bit wide elements (unpredicated).

ASRD: Arithmetic shift right for divide by immediate (predicated).

ASRR: Reversed arithmetic shift right by vector (predicated).

BCAX: Bitwise clear and exclusive OR.

BDEP: Scatter lower bits into positions selected by bitmask.

BEXT: Gather lower bits from positions selected by bitmask.

BFCVT: Floating-point down convert to BFloat16 format (predicated).

BFCVTNT: Floating-point down convert and narrow to BFloat16 (top, predicated).

BFDOT (indexed): BFloat16 floating-point indexed dot product.

BFDOT (vectors): BFloat16 floating-point dot product.

BFMLALB (indexed): BFloat16 floating-point multiply-add long to single-precision (bottom, indexed).

BFMLALB (vectors): BFloat16 floating-point multiply-add long to single-precision (bottom).

BFMLALT (indexed): BFloat16 floating-point multiply-add long to single-precision (top, indexed).

BFMLALT (vectors): BFloat16 floating-point multiply-add long to single-precision (top).

BFMMLA: BFloat16 floating-point matrix multiply-accumulate.

BGRP: Group bits to right or left as selected by bitmask.

BIC (immediate): Bitwise clear bits using immediate (unpredicated): an alias of AND (immediate).

BIC (predicates): Bitwise clear predicates.

BIC (vectors, predicated): Bitwise clear vectors (predicated).

BIC (vectors, unpredicated): Bitwise clear vectors (unpredicated).

BICS: Bitwise clear predicates, setting the condition flags.

BRKA: Break after first true condition.

BRKAS: Break after first true condition, setting the condition flags.

BRKB: Break before first true condition.

BRKBS: Break before first true condition, setting the condition flags.

BRKN: Propagate break to next partition.

BRKNS: Propagate break to next partition, setting the condition flags.

BRKPA: Break after first true condition, propagating from previous partition.

BRKPAS: Break after first true condition, propagating from previous partition and setting the condition flags.

BRKPB: Break before first true condition, propagating from previous partition.

BRKPBS: Break before first true condition, propagating from previous partition and setting the condition flags.

BSL: Bitwise select.

BSL1N: Bitwise select with first input inverted.

BSL2N: Bitwise select with second input inverted.

CADD: Complex integer add with rotate.

CDOT (indexed): Complex integer dot product (indexed).

CDOT (vectors): Complex integer dot product.

CLASTA (scalar): Conditionally extract element after last to general-purpose register.

CLASTA (SIMD&FP scalar): Conditionally extract element after last to SIMD&FP scalar register.

CLASTA (vectors): Conditionally extract element after last to vector register.

CLASTB (scalar): Conditionally extract last element to general-purpose register.

CLASTB (SIMD&FP scalar): Conditionally extract last element to SIMD&FP scalar register.

CLASTB (vectors): Conditionally extract last element to vector register.

CLS: Count leading sign bits (predicated).

CLZ: Count leading zero bits (predicated).

CMLA (indexed): Complex integer multiply-add with rotate (indexed).

CMLA (vectors): Complex integer multiply-add with rotate.

CMP<cc> (immediate): Compare vector to immediate.

CMP<cc> (vectors): Compare vectors.

CMP<cc> (wide elements): Compare vector to 64-bit wide elements.

CMPLE (vectors): Compare signed less than or equal to vector, setting the condition flags: an alias of CMP<cc> (vectors).

CMPLO (vectors): Compare unsigned lower than vector, setting the condition flags: an alias of CMP<cc> (vectors).

CMPLS (vectors): Compare unsigned lower or same as vector, setting the condition flags: an alias of CMP<cc> (vectors).

CMPLT (vectors): Compare signed less than vector, setting the condition flags: an alias of CMP<cc> (vectors).

CNOT: Logically invert boolean condition in vector (predicated).

CNT: Count non-zero bits (predicated).

CNTB, CNTD, CNTH, CNTW: Set scalar to multiple of predicate constraint element count.

CNTP: Set scalar to count of true predicate elements.

COMPACT: Shuffle active elements of vector to the right and fill with zero.

CPY (immediate, merging): Copy signed integer immediate to vector elements (merging).

CPY (immediate, zeroing): Copy signed integer immediate to vector elements (zeroing).

CPY (scalar): Copy general-purpose register to vector elements (predicated).

CPY (SIMD&FP scalar): Copy SIMD&FP scalar register to vector elements (predicated).

CTERMEQ, CTERMNE: Compare and terminate loop.

DECB, DECD, DECH, DECW (scalar): Decrement scalar by multiple of predicate constraint element count.

DECD, DECH, DECW (vector): Decrement vector by multiple of predicate constraint element count.

DECP (scalar): Decrement scalar by count of true predicate elements.

DECP (vector): Decrement vector by count of true predicate elements.

DUP (immediate): Broadcast signed immediate to vector elements (unpredicated).

DUP (indexed): Broadcast indexed element to vector (unpredicated).

DUP (scalar): Broadcast general-purpose register to vector elements (unpredicated).

DUPM: Broadcast logical bitmask immediate to vector (unpredicated).

EON: Bitwise exclusive OR with inverted immediate (unpredicated): an alias of EOR (immediate).

EOR (immediate): Bitwise exclusive OR with immediate (unpredicated).

EOR (predicates): Bitwise exclusive OR predicates.

EOR (vectors, predicated): Bitwise exclusive OR vectors (predicated).

EOR (vectors, unpredicated): Bitwise exclusive OR vectors (unpredicated).

EOR3: Bitwise exclusive OR of three vectors.

EORBT: Interleaving exclusive OR (bottom, top).

EORS: Bitwise exclusive OR predicates, setting the condition flags.

EORTB: Interleaving exclusive OR (top, bottom).

EORV: Bitwise exclusive OR reduction to scalar.

EXT: Extract vector from pair of vectors.

FABD: Floating-point absolute difference (predicated).

FABS: Floating-point absolute value (predicated).

FAC<cc>: Floating-point absolute compare vectors.

FACLE: Floating-point absolute compare less than or equal: an alias of FAC<cc>.

FACLT: Floating-point absolute compare less than: an alias of FAC<cc>.

FADD (immediate): Floating-point add immediate (predicated).

FADD (vectors, predicated): Floating-point add vector (predicated).

FADD (vectors, unpredicated): Floating-point add vector (unpredicated).

FADDA: Floating-point add strictly-ordered reduction, accumulating in scalar.

FADDP: Floating-point add pairwise.

FADDV: Floating-point add recursive reduction to scalar.

FCADD: Floating-point complex add with rotate (predicated).

FCM<cc> (vectors): Floating-point compare vectors.

FCM<cc> (zero): Floating-point compare vector with zero.

FCMLA (indexed): Floating-point complex multiply-add by indexed values with rotate.

FCMLA (vectors): Floating-point complex multiply-add with rotate (predicated).

FCMLE (vectors): Floating-point compare less than or equal to vector: an alias of FCM<cc> (vectors).

FCMLT (vectors): Floating-point compare less than vector: an alias of FCM<cc> (vectors).

FCPY: Copy 8-bit floating-point immediate to vector elements (predicated).

FCVT: Floating-point convert precision (predicated).

FCVTLT: Floating-point up convert long (top, predicated).

FCVTNT: Floating-point down convert and narrow (top, predicated).

FCVTX: Floating-point down convert, rounding to odd (predicated).

FCVTXNT: Floating-point down convert, rounding to odd (top, predicated).

FCVTZS: Floating-point convert to signed integer, rounding toward zero (predicated).

FCVTZU: Floating-point convert to unsigned integer, rounding toward zero (predicated).

FDIV: Floating-point divide by vector (predicated).

FDIVR: Floating-point reversed divide by vector (predicated).

FDUP: Broadcast 8-bit floating-point immediate to vector elements (unpredicated).

FEXPA: Floating-point exponential accelerator.

FLOGB: Floating-point base 2 logarithm as integer.

FMAD: Floating-point fused multiply-add vectors (predicated), writing multiplicand [Zdn = Za + Zdn * Zm].

FMAX (immediate): Floating-point maximum with immediate (predicated).

FMAX (vectors): Floating-point maximum (predicated).

FMAXNM (immediate): Floating-point maximum number with immediate (predicated).

FMAXNM (vectors): Floating-point maximum number (predicated).

FMAXNMP: Floating-point maximum number pairwise.

FMAXNMV: Floating-point maximum number recursive reduction to scalar.

FMAXP: Floating-point maximum pairwise.

FMAXV: Floating-point maximum recursive reduction to scalar.

FMIN (immediate): Floating-point minimum with immediate (predicated).

FMIN (vectors): Floating-point minimum (predicated).

FMINNM (immediate): Floating-point minimum number with immediate (predicated).

FMINNM (vectors): Floating-point minimum number (predicated).

FMINNMP: Floating-point minimum number pairwise.

FMINNMV: Floating-point minimum number recursive reduction to scalar.

FMINP: Floating-point minimum pairwise.

FMINV: Floating-point minimum recursive reduction to scalar.

FMLA (indexed): Floating-point fused multiply-add by indexed elements (Zda = Zda + Zn * Zm[indexed]).

FMLA (vectors): Floating-point fused multiply-add vectors (predicated), writing addend [Zda = Zda + Zn * Zm].

FMLALB (indexed): Half-precision floating-point multiply-add long to single-precision (bottom, indexed).

FMLALB (vectors): Half-precision floating-point multiply-add long to single-precision (bottom).

FMLALT (indexed): Half-precision floating-point multiply-add long to single-precision (top, indexed).

FMLALT (vectors): Half-precision floating-point multiply-add long to single-precision (top).

FMLS (indexed): Floating-point fused multiply-subtract by indexed elements (Zda = Zda + -Zn * Zm[indexed]).

FMLS (vectors): Floating-point fused multiply-subtract vectors (predicated), writing addend [Zda = Zda + -Zn * Zm].

FMLSLB (indexed): Half-precision floating-point multiply-subtract long from single-precision (bottom, indexed).

FMLSLB (vectors): Half-precision floating-point multiply-subtract long from single-precision (bottom).

FMLSLT (indexed): Half-precision floating-point multiply-subtract long from single-precision (top, indexed).

FMLSLT (vectors): Half-precision floating-point multiply-subtract long from single-precision (top).

FMMLA: Floating-point matrix multiply-accumulate.

FMOV (immediate, predicated): Move 8-bit floating-point immediate to vector elements (predicated): an alias of FCPY.

FMOV (immediate, unpredicated): Move 8-bit floating-point immediate to vector elements (unpredicated): an alias of FDUP.

FMOV (zero, predicated): Move floating-point +0.0 to vector elements (predicated): an alias of CPY (immediate, merging).

FMOV (zero, unpredicated): Move floating-point +0.0 to vector elements (unpredicated): an alias of DUP (immediate).

FMSB: Floating-point fused multiply-subtract vectors (predicated), writing multiplicand [Zdn = Za + -Zdn * Zm].

FMUL (immediate): Floating-point multiply by immediate (predicated).

FMUL (indexed): Floating-point multiply by indexed elements.

FMUL (vectors, predicated): Floating-point multiply vectors (predicated).

FMUL (vectors, unpredicated): Floating-point multiply vectors (unpredicated).

FMULX: Floating-point multiply-extended vectors (predicated).

FNEG: Floating-point negate (predicated).

FNMAD: Floating-point negated fused multiply-add vectors (predicated), writing multiplicand [Zdn = -Za + -Zdn * Zm].

FNMLA: Floating-point negated fused multiply-add vectors (predicated), writing addend [Zda = -Zda + -Zn * Zm].

FNMLS: Floating-point negated fused multiply-subtract vectors (predicated), writing addend [Zda = -Zda + Zn * Zm].

FNMSB: Floating-point negated fused multiply-subtract vectors (predicated), writing multiplicand [Zdn = -Za + Zdn * Zm].

FRECPE: Floating-point reciprocal estimate (unpredicated).

FRECPS: Floating-point reciprocal step (unpredicated).

FRECPX: Floating-point reciprocal exponent (predicated).

FRINT<r>: Floating-point round to integral value (predicated).

FRSQRTE: Floating-point reciprocal square root estimate (unpredicated).

FRSQRTS: Floating-point reciprocal square root step (unpredicated).

FSCALE: Floating-point adjust exponent by vector (predicated).

FSQRT: Floating-point square root (predicated).

FSUB (immediate): Floating-point subtract immediate (predicated).

FSUB (vectors, predicated): Floating-point subtract vectors (predicated).

FSUB (vectors, unpredicated): Floating-point subtract vectors (unpredicated).

FSUBR (immediate): Floating-point reversed subtract from immediate (predicated).

FSUBR (vectors): Floating-point reversed subtract vectors (predicated).

FTMAD: Floating-point trigonometric multiply-add coefficient.

FTSMUL: Floating-point trigonometric starting value.

FTSSEL: Floating-point trigonometric select coefficient.

HISTCNT: Count matching elements in vector.

HISTSEG: Count matching elements in vector segments.

INCB, INCD, INCH, INCW (scalar): Increment scalar by multiple of predicate constraint element count.

INCD, INCH, INCW (vector): Increment vector by multiple of predicate constraint element count.

INCP (scalar): Increment scalar by count of true predicate elements.

INCP (vector): Increment vector by count of true predicate elements.

INDEX (immediate, scalar): Create index starting from immediate and incremented by general-purpose register.

INDEX (immediates): Create index starting from and incremented by immediate.

INDEX (scalar, immediate): Create index starting from general-purpose register and incremented by immediate.

INDEX (scalars): Create index starting from and incremented by general-purpose register.

INSR (scalar): Insert general-purpose register in shifted vector.

INSR (SIMD&FP scalar): Insert SIMD&FP scalar register in shifted vector.

LASTA (scalar): Extract element after last to general-purpose register.

LASTA (SIMD&FP scalar): Extract element after last to SIMD&FP scalar register.

LASTB (scalar): Extract last element to general-purpose register.

LASTB (SIMD&FP scalar): Extract last element to SIMD&FP scalar register.

LD1B (scalar plus immediate): Contiguous load unsigned bytes to vector (immediate index).

LD1B (scalar plus scalar): Contiguous load unsigned bytes to vector (scalar index).

LD1B (scalar plus vector): Gather load unsigned bytes to vector (vector index).

LD1B (vector plus immediate): Gather load unsigned bytes to vector (immediate index).

LD1D (scalar plus immediate): Contiguous load doublewords to vector (immediate index).

LD1D (scalar plus scalar): Contiguous load doublewords to vector (scalar index).

LD1D (scalar plus vector): Gather load doublewords to vector (vector index).

LD1D (vector plus immediate): Gather load doublewords to vector (immediate index).

LD1H (scalar plus immediate): Contiguous load unsigned halfwords to vector (immediate index).

LD1H (scalar plus scalar): Contiguous load unsigned halfwords to vector (scalar index).

LD1H (scalar plus vector): Gather load unsigned halfwords to vector (vector index).

LD1H (vector plus immediate): Gather load unsigned halfwords to vector (immediate index).

LD1RB: Load and broadcast unsigned byte to vector.

LD1RD: Load and broadcast doubleword to vector.

LD1RH: Load and broadcast unsigned halfword to vector.

LD1ROB (scalar plus immediate): Contiguous load and replicate thirty-two bytes (immediate index).

LD1ROB (scalar plus scalar): Contiguous load and replicate thirty-two bytes (scalar index).

LD1ROD (scalar plus immediate): Contiguous load and replicate four doublewords (immediate index).

LD1ROD (scalar plus scalar): Contiguous load and replicate four doublewords (scalar index).

LD1ROH (scalar plus immediate): Contiguous load and replicate sixteen halfwords (immediate index).

LD1ROH (scalar plus scalar): Contiguous load and replicate sixteen halfwords (scalar index).

LD1ROW (scalar plus immediate): Contiguous load and replicate eight words (immediate index).

LD1ROW (scalar plus scalar): Contiguous load and replicate eight words (scalar index).

LD1RQB (scalar plus immediate): Contiguous load and replicate sixteen bytes (immediate index).

LD1RQB (scalar plus scalar): Contiguous load and replicate sixteen bytes (scalar index).

LD1RQD (scalar plus immediate): Contiguous load and replicate two doublewords (immediate index).

LD1RQD (scalar plus scalar): Contiguous load and replicate two doublewords (scalar index).

LD1RQH (scalar plus immediate): Contiguous load and replicate eight halfwords (immediate index).

LD1RQH (scalar plus scalar): Contiguous load and replicate eight halfwords (scalar index).

LD1RQW (scalar plus immediate): Contiguous load and replicate four words (immediate index).

LD1RQW (scalar plus scalar): Contiguous load and replicate four words (scalar index).

LD1RSB: Load and broadcast signed byte to vector.

LD1RSH: Load and broadcast signed halfword to vector.

LD1RSW: Load and broadcast signed word to vector.

LD1RW: Load and broadcast unsigned word to vector.

LD1SB (scalar plus immediate): Contiguous load signed bytes to vector (immediate index).

LD1SB (scalar plus scalar): Contiguous load signed bytes to vector (scalar index).

LD1SB (scalar plus vector): Gather load signed bytes to vector (vector index).

LD1SB (vector plus immediate): Gather load signed bytes to vector (immediate index).

LD1SH (scalar plus immediate): Contiguous load signed halfwords to vector (immediate index).

LD1SH (scalar plus scalar): Contiguous load signed halfwords to vector (scalar index).

LD1SH (scalar plus vector): Gather load signed halfwords to vector (vector index).

LD1SH (vector plus immediate): Gather load signed halfwords to vector (immediate index).

LD1SW (scalar plus immediate): Contiguous load signed words to vector (immediate index).

LD1SW (scalar plus scalar): Contiguous load signed words to vector (scalar index).

LD1SW (scalar plus vector): Gather load signed words to vector (vector index).

LD1SW (vector plus immediate): Gather load signed words to vector (immediate index).

LD1W (scalar plus immediate): Contiguous load unsigned words to vector (immediate index).

LD1W (scalar plus scalar): Contiguous load unsigned words to vector (scalar index).

LD1W (scalar plus vector): Gather load unsigned words to vector (vector index).

LD1W (vector plus immediate): Gather load unsigned words to vector (immediate index).

LD2B (scalar plus immediate): Contiguous load two-byte structures to two vectors (immediate index).

LD2B (scalar plus scalar): Contiguous load two-byte structures to two vectors (scalar index).

LD2D (scalar plus immediate): Contiguous load two-doubleword structures to two vectors (immediate index).

LD2D (scalar plus scalar): Contiguous load two-doubleword structures to two vectors (scalar index).

LD2H (scalar plus immediate): Contiguous load two-halfword structures to two vectors (immediate index).

LD2H (scalar plus scalar): Contiguous load two-halfword structures to two vectors (scalar index).

LD2W (scalar plus immediate): Contiguous load two-word structures to two vectors (immediate index).

LD2W (scalar plus scalar): Contiguous load two-word structures to two vectors (scalar index).

LD3B (scalar plus immediate): Contiguous load three-byte structures to three vectors (immediate index).

LD3B (scalar plus scalar): Contiguous load three-byte structures to three vectors (scalar index).

LD3D (scalar plus immediate): Contiguous load three-doubleword structures to three vectors (immediate index).

LD3D (scalar plus scalar): Contiguous load three-doubleword structures to three vectors (scalar index).

LD3H (scalar plus immediate): Contiguous load three-halfword structures to three vectors (immediate index).

LD3H (scalar plus scalar): Contiguous load three-halfword structures to three vectors (scalar index).

LD3W (scalar plus immediate): Contiguous load three-word structures to three vectors (immediate index).

LD3W (scalar plus scalar): Contiguous load three-word structures to three vectors (scalar index).

LD4B (scalar plus immediate): Contiguous load four-byte structures to four vectors (immediate index).

LD4B (scalar plus scalar): Contiguous load four-byte structures to four vectors (scalar index).

LD4D (scalar plus immediate): Contiguous load four-doubleword structures to four vectors (immediate index).

LD4D (scalar plus scalar): Contiguous load four-doubleword structures to four vectors (scalar index).

LD4H (scalar plus immediate): Contiguous load four-halfword structures to four vectors (immediate index).

LD4H (scalar plus scalar): Contiguous load four-halfword structures to four vectors (scalar index).

LD4W (scalar plus immediate): Contiguous load four-word structures to four vectors (immediate index).

LD4W (scalar plus scalar): Contiguous load four-word structures to four vectors (scalar index).

LDFF1B (scalar plus scalar): Contiguous load first-fault unsigned bytes to vector (scalar index).

LDFF1B (scalar plus vector): Gather load first-fault unsigned bytes to vector (vector index).

LDFF1B (vector plus immediate): Gather load first-fault unsigned bytes to vector (immediate index).

LDFF1D (scalar plus scalar): Contiguous load first-fault doublewords to vector (scalar index).

LDFF1D (scalar plus vector): Gather load first-fault doublewords to vector (vector index).

LDFF1D (vector plus immediate): Gather load first-fault doublewords to vector (immediate index).

LDFF1H (scalar plus scalar): Contiguous load first-fault unsigned halfwords to vector (scalar index).

LDFF1H (scalar plus vector): Gather load first-fault unsigned halfwords to vector (vector index).

LDFF1H (vector plus immediate): Gather load first-fault unsigned halfwords to vector (immediate index).

LDFF1SB (scalar plus scalar): Contiguous load first-fault signed bytes to vector (scalar index).

LDFF1SB (scalar plus vector): Gather load first-fault signed bytes to vector (vector index).

LDFF1SB (vector plus immediate): Gather load first-fault signed bytes to vector (immediate index).

LDFF1SH (scalar plus scalar): Contiguous load first-fault signed halfwords to vector (scalar index).

LDFF1SH (scalar plus vector): Gather load first-fault signed halfwords to vector (vector index).

LDFF1SH (vector plus immediate): Gather load first-fault signed halfwords to vector (immediate index).

LDFF1SW (scalar plus scalar): Contiguous load first-fault signed words to vector (scalar index).

LDFF1SW (scalar plus vector): Gather load first-fault signed words to vector (vector index).

LDFF1SW (vector plus immediate): Gather load first-fault signed words to vector (immediate index).

LDFF1W (scalar plus scalar): Contiguous load first-fault unsigned words to vector (scalar index).

LDFF1W (scalar plus vector): Gather load first-fault unsigned words to vector (vector index).

LDFF1W (vector plus immediate): Gather load first-fault unsigned words to vector (immediate index).

LDNF1B: Contiguous load non-fault unsigned bytes to vector (immediate index).

LDNF1D: Contiguous load non-fault doublewords to vector (immediate index).

LDNF1H: Contiguous load non-fault unsigned halfwords to vector (immediate index).

LDNF1SB: Contiguous load non-fault signed bytes to vector (immediate index).

LDNF1SH: Contiguous load non-fault signed halfwords to vector (immediate index).

LDNF1SW: Contiguous load non-fault signed words to vector (immediate index).

LDNF1W: Contiguous load non-fault unsigned words to vector (immediate index).

LDNT1B (scalar plus immediate): Contiguous load non-temporal bytes to vector (immediate index).

LDNT1B (scalar plus scalar): Contiguous load non-temporal bytes to vector (scalar index).

LDNT1B (vector plus scalar): Gather load non-temporal unsigned bytes.

LDNT1D (scalar plus immediate): Contiguous load non-temporal doublewords to vector (immediate index).

LDNT1D (scalar plus scalar): Contiguous load non-temporal doublewords to vector (scalar index).

LDNT1D (vector plus scalar): Gather load non-temporal unsigned doublewords.

LDNT1H (scalar plus immediate): Contiguous load non-temporal halfwords to vector (immediate index).

LDNT1H (scalar plus scalar): Contiguous load non-temporal halfwords to vector (scalar index).

LDNT1H (vector plus scalar): Gather load non-temporal unsigned halfwords.

LDNT1SB: Gather load non-temporal signed bytes.

LDNT1SH: Gather load non-temporal signed halfwords.

LDNT1SW: Gather load non-temporal signed words.

LDNT1W (scalar plus immediate): Contiguous load non-temporal words to vector (immediate index).

LDNT1W (scalar plus scalar): Contiguous load non-temporal words to vector (scalar index).

LDNT1W (vector plus scalar): Gather load non-temporal unsigned words.

LDR (predicate): Load predicate register.

LDR (vector): Load vector register.

LSL (immediate, predicated): Logical shift left by immediate (predicated).

LSL (immediate, unpredicated): Logical shift left by immediate (unpredicated).

LSL (vectors): Logical shift left by vector (predicated).

LSL (wide elements, predicated): Logical shift left by 64-bit wide elements (predicated).

LSL (wide elements, unpredicated): Logical shift left by 64-bit wide elements (unpredicated).

LSLR: Reversed logical shift left by vector (predicated).

LSR (immediate, predicated): Logical shift right by immediate (predicated).

LSR (immediate, unpredicated): Logical shift right by immediate (unpredicated).

LSR (vectors): Logical shift right by vector (predicated).

LSR (wide elements, predicated): Logical shift right by 64-bit wide elements (predicated).

LSR (wide elements, unpredicated): Logical shift right by 64-bit wide elements (unpredicated).

LSRR: Reversed logical shift right by vector (predicated).

MAD: Multiply-add vectors (predicated), writing multiplicand [Zdn = Za + Zdn * Zm].

MATCH: Detect any matching elements, setting the condition flags.

MLA (indexed): Multiply-add to accumulator (indexed).

MLA (vectors): Multiply-add vectors (predicated), writing addend [Zda = Zda + Zn * Zm].

MLS (indexed): Multiply-subtract from accumulator (indexed).

MLS (vectors): Multiply-subtract vectors (predicated), writing addend [Zda = Zda - Zn * Zm].

MOV: Move logical bitmask immediate to vector (unpredicated): an alias of DUPM.

MOV: Move predicate (unpredicated): an alias of ORR (predicates).

MOV (immediate, predicated, merging): Move signed integer immediate to vector elements (merging): an alias of CPY (immediate, merging).

MOV (immediate, predicated, zeroing): Move signed integer immediate to vector elements (zeroing): an alias of CPY (immediate, zeroing).

MOV (immediate, unpredicated): Move signed immediate to vector elements (unpredicated): an alias of DUP (immediate).

MOV (predicate, predicated, merging): Move predicates (merging): an alias of SEL (predicates).

MOV (predicate, predicated, zeroing): Move predicates (zeroing): an alias of AND (predicates).

MOV (scalar, predicated): Move general-purpose register to vector elements (predicated): an alias of CPY (scalar).

MOV (scalar, unpredicated): Move general-purpose register to vector elements (unpredicated): an alias of DUP (scalar).

MOV (SIMD&FP scalar, predicated): Move SIMD&FP scalar register to vector elements (predicated): an alias of CPY (SIMD&FP scalar).

MOV (SIMD&FP scalar, unpredicated): Move indexed element or SIMD&FP scalar to vector (unpredicated): an alias of DUP (indexed).

MOV (vector, predicated): Move vector elements (predicated): an alias of SEL (vectors).

MOV (vector, unpredicated): Move vector register (unpredicated): an alias of ORR (vectors, unpredicated).

MOVPRFX (predicated): Move prefix (predicated).

MOVPRFX (unpredicated): Move prefix (unpredicated).

MOVS (predicated): Move predicates (zeroing), setting the condition flags: an alias of ANDS.

MOVS (unpredicated): Move predicate (unpredicated), setting the condition flags: an alias of ORRS.

MSB: Multiply-subtract vectors (predicated), writing multiplicand [Zdn = Za - Zdn * Zm].

MUL (immediate): Multiply by immediate (unpredicated).

MUL (indexed): Multiply (indexed).

MUL (vectors, predicated): Multiply vectors (predicated).

MUL (vectors, unpredicated): Multiply vectors (unpredicated).

NAND: Bitwise NAND predicates.

NANDS: Bitwise NAND predicates, setting the condition flags.

NBSL: Bitwise inverted select.

NEG: Negate (predicated).

NMATCH: Detect no matching elements, setting the condition flags.

NOR: Bitwise NOR predicates.

NORS: Bitwise NOR predicates, setting the condition flags.

NOT (predicate): Bitwise invert predicate: an alias of EOR (predicates).

NOT (vector): Bitwise invert vector (predicated).

NOTS: Bitwise invert predicate, setting the condition flags: an alias of EORS.

ORN (immediate): Bitwise inclusive OR with inverted immediate (unpredicated): an alias of ORR (immediate).

ORN (predicates): Bitwise inclusive OR inverted predicate.

ORNS: Bitwise inclusive OR inverted predicate, setting the condition flags.

ORR (immediate): Bitwise inclusive OR with immediate (unpredicated).

ORR (predicates): Bitwise inclusive OR predicates.

ORR (vectors, predicated): Bitwise inclusive OR vectors (predicated).

ORR (vectors, unpredicated): Bitwise inclusive OR vectors (unpredicated).

ORRS: Bitwise inclusive OR predicates, setting the condition flags.

ORV: Bitwise inclusive OR reduction to scalar.

PFALSE: Set all predicate elements to false.

PFIRST: Set the first active predicate element to true.

PMUL: Polynomial multiply vectors (unpredicated).

PMULLB: Polynomial multiply long (bottom).

PMULLT: Polynomial multiply long (top).

PNEXT: Find next active predicate.

PRFB (scalar plus immediate): Contiguous prefetch bytes (immediate index).

PRFB (scalar plus scalar): Contiguous prefetch bytes (scalar index).

PRFB (scalar plus vector): Gather prefetch bytes (scalar plus vector).

PRFB (vector plus immediate): Gather prefetch bytes (vector plus immediate).

PRFD (scalar plus immediate): Contiguous prefetch doublewords (immediate index).

PRFD (scalar plus scalar): Contiguous prefetch doublewords (scalar index).

PRFD (scalar plus vector): Gather prefetch doublewords (scalar plus vector).

PRFD (vector plus immediate): Gather prefetch doublewords (vector plus immediate).

PRFH (scalar plus immediate): Contiguous prefetch halfwords (immediate index).

PRFH (scalar plus scalar): Contiguous prefetch halfwords (scalar index).

PRFH (scalar plus vector): Gather prefetch halfwords (scalar plus vector).

PRFH (vector plus immediate): Gather prefetch halfwords (vector plus immediate).

PRFW (scalar plus immediate): Contiguous prefetch words (immediate index).

PRFW (scalar plus scalar): Contiguous prefetch words (scalar index).

PRFW (scalar plus vector): Gather prefetch words (scalar plus vector).

PRFW (vector plus immediate): Gather prefetch words (vector plus immediate).

PSEL: Predicate select between predicate register or all-false.

PTEST: Set condition flags for predicate.

PTRUE: Initialise predicate from named constraint.

PTRUES: Initialise predicate from named constraint and set the condition flags.

PUNPKHI, PUNPKLO: Unpack and widen half of predicate.

RADDHNB: Rounding add narrow high part (bottom).

RADDHNT: Rounding add narrow high part (top).

RAX1: Bitwise rotate left by 1 and exclusive OR.

RBIT: Reverse bits (predicated).

RDFFR (predicated): Return predicate of succesfully loaded elements.

RDFFR (unpredicated): Read the first-fault register.

RDFFRS: Return predicate of succesfully loaded elements, setting the condition flags.

RDVL: Read multiple of vector register size to scalar register.

REV (predicate): Reverse all elements in a predicate.

REV (vector): Reverse all elements in a vector (unpredicated).

REVB, REVH, REVW: Reverse bytes / halfwords / words within elements (predicated).

REVD: Reverse 64-bit doublewords in elements (predicated).

RSHRNB: Rounding shift right narrow by immediate (bottom).

RSHRNT: Rounding shift right narrow by immediate (top).

RSUBHNB: Rounding subtract narrow high part (bottom).

RSUBHNT: Rounding subtract narrow high part (top).

SABA: Signed absolute difference and accumulate.

SABALB: Signed absolute difference and accumulate long (bottom).

SABALT: Signed absolute difference and accumulate long (top).

SABD: Signed absolute difference (predicated).

SABDLB: Signed absolute difference long (bottom).

SABDLT: Signed absolute difference long (top).

SADALP: Signed add and accumulate long pairwise.

SADDLB: Signed add long (bottom).

SADDLBT: Signed add long (bottom + top).

SADDLT: Signed add long (top).

SADDV: Signed add reduction to scalar.

SADDWB: Signed add wide (bottom).

SADDWT: Signed add wide (top).

SBCLB: Subtract with carry long (bottom).

SBCLT: Subtract with carry long (top).

SCLAMP: Signed clamp to minimum/maximum vector.

SCVTF: Signed integer convert to floating-point (predicated).

SDIV: Signed divide (predicated).

SDIVR: Signed reversed divide (predicated).

SDOT (indexed): Signed integer indexed dot product.

SDOT (vectors): Signed integer dot product.

SEL (predicates): Conditionally select elements from two predicates.

SEL (vectors): Conditionally select elements from two vectors.

SETFFR: Initialise the first-fault register to all true.

SHADD: Signed halving addition.

SHRNB: Shift right narrow by immediate (bottom).

SHRNT: Shift right narrow by immediate (top).

SHSUB: Signed halving subtract.

SHSUBR: Signed halving subtract reversed vectors.

SLI: Shift left and insert (immediate).

SM4E: SM4 encryption and decryption.

SM4EKEY: SM4 key updates.

SMAX (immediate): Signed maximum with immediate (unpredicated).

SMAX (vectors): Signed maximum vectors (predicated).

SMAXP: Signed maximum pairwise.

SMAXV: Signed maximum reduction to scalar.

SMIN (immediate): Signed minimum with immediate (unpredicated).

SMIN (vectors): Signed minimum vectors (predicated).

SMINP: Signed minimum pairwise.

SMINV: Signed minimum reduction to scalar.

SMLALB (indexed): Signed multiply-add long to accumulator (bottom, indexed).

SMLALB (vectors): Signed multiply-add long to accumulator (bottom).

SMLALT (indexed): Signed multiply-add long to accumulator (top, indexed).

SMLALT (vectors): Signed multiply-add long to accumulator (top).

SMLSLB (indexed): Signed multiply-subtract long from accumulator (bottom, indexed).

SMLSLB (vectors): Signed multiply-subtract long from accumulator (bottom).

SMLSLT (indexed): Signed multiply-subtract long from accumulator (top, indexed).

SMLSLT (vectors): Signed multiply-subtract long from accumulator (top).

SMMLA: Signed integer matrix multiply-accumulate.

SMULH (predicated): Signed multiply returning high half (predicated).

SMULH (unpredicated): Signed multiply returning high half (unpredicated).

SMULLB (indexed): Signed multiply long (bottom, indexed).

SMULLB (vectors): Signed multiply long (bottom).

SMULLT (indexed): Signed multiply long (top, indexed).

SMULLT (vectors): Signed multiply long (top).

SPLICE: Splice two vectors under predicate control.

SQABS: Signed saturating absolute value.

SQADD (immediate): Signed saturating add immediate (unpredicated).

SQADD (vectors, predicated): Signed saturating addition (predicated).

SQADD (vectors, unpredicated): Signed saturating add vectors (unpredicated).

SQCADD: Saturating complex integer add with rotate.

SQDECB: Signed saturating decrement scalar by multiple of 8-bit predicate constraint element count.

SQDECD (scalar): Signed saturating decrement scalar by multiple of 64-bit predicate constraint element count.

SQDECD (vector): Signed saturating decrement vector by multiple of 64-bit predicate constraint element count.

SQDECH (scalar): Signed saturating decrement scalar by multiple of 16-bit predicate constraint element count.

SQDECH (vector): Signed saturating decrement vector by multiple of 16-bit predicate constraint element count.

SQDECP (scalar): Signed saturating decrement scalar by count of true predicate elements.

SQDECP (vector): Signed saturating decrement vector by count of true predicate elements.

SQDECW (scalar): Signed saturating decrement scalar by multiple of 32-bit predicate constraint element count.

SQDECW (vector): Signed saturating decrement vector by multiple of 32-bit predicate constraint element count.

SQDMLALB (indexed): Signed saturating doubling multiply-add long to accumulator (bottom, indexed).

SQDMLALB (vectors): Signed saturating doubling multiply-add long to accumulator (bottom).

SQDMLALBT: Signed saturating doubling multiply-add long to accumulator (bottom × top).

SQDMLALT (indexed): Signed saturating doubling multiply-add long to accumulator (top, indexed).

SQDMLALT (vectors): Signed saturating doubling multiply-add long to accumulator (top).

SQDMLSLB (indexed): Signed saturating doubling multiply-subtract long from accumulator (bottom, indexed).

SQDMLSLB (vectors): Signed saturating doubling multiply-subtract long from accumulator (bottom).

SQDMLSLBT: Signed saturating doubling multiply-subtract long from accumulator (bottom × top).

SQDMLSLT (indexed): Signed saturating doubling multiply-subtract long from accumulator (top, indexed).

SQDMLSLT (vectors): Signed saturating doubling multiply-subtract long from accumulator (top).

SQDMULH (indexed): Signed saturating doubling multiply high (indexed).

SQDMULH (vectors): Signed saturating doubling multiply high (unpredicated).

SQDMULLB (indexed): Signed saturating doubling multiply long (bottom, indexed).

SQDMULLB (vectors): Signed saturating doubling multiply long (bottom).

SQDMULLT (indexed): Signed saturating doubling multiply long (top, indexed).

SQDMULLT (vectors): Signed saturating doubling multiply long (top).

SQINCB: Signed saturating increment scalar by multiple of 8-bit predicate constraint element count.

SQINCD (scalar): Signed saturating increment scalar by multiple of 64-bit predicate constraint element count.

SQINCD (vector): Signed saturating increment vector by multiple of 64-bit predicate constraint element count.

SQINCH (scalar): Signed saturating increment scalar by multiple of 16-bit predicate constraint element count.

SQINCH (vector): Signed saturating increment vector by multiple of 16-bit predicate constraint element count.

SQINCP (scalar): Signed saturating increment scalar by count of true predicate elements.

SQINCP (vector): Signed saturating increment vector by count of true predicate elements.

SQINCW (scalar): Signed saturating increment scalar by multiple of 32-bit predicate constraint element count.

SQINCW (vector): Signed saturating increment vector by multiple of 32-bit predicate constraint element count.

SQNEG: Signed saturating negate.

SQRDCMLAH (indexed): Saturating rounding doubling complex integer multiply-add high with rotate (indexed).

SQRDCMLAH (vectors): Saturating rounding doubling complex integer multiply-add high with rotate.

SQRDMLAH (indexed): Signed saturating rounding doubling multiply-add high to accumulator (indexed).

SQRDMLAH (vectors): Signed saturating rounding doubling multiply-add high to accumulator (unpredicated).

SQRDMLSH (indexed): Signed saturating rounding doubling multiply-subtract high from accumulator (indexed).

SQRDMLSH (vectors): Signed saturating rounding doubling multiply-subtract high from accumulator (unpredicated).

SQRDMULH (indexed): Signed saturating rounding doubling multiply high (indexed).

SQRDMULH (vectors): Signed saturating rounding doubling multiply high (unpredicated).

SQRSHL: Signed saturating rounding shift left by vector (predicated).

SQRSHLR: Signed saturating rounding shift left reversed vectors (predicated).

SQRSHRNB: Signed saturating rounding shift right narrow by immediate (bottom).

SQRSHRNT: Signed saturating rounding shift right narrow by immediate (top).

SQRSHRUNB: Signed saturating rounding shift right unsigned narrow by immediate (bottom).

SQRSHRUNT: Signed saturating rounding shift right unsigned narrow by immediate (top).

SQSHL (immediate): Signed saturating shift left by immediate.

SQSHL (vectors): Signed saturating shift left by vector (predicated).

SQSHLR: Signed saturating shift left reversed vectors (predicated).

SQSHLU: Signed saturating shift left unsigned by immediate.

SQSHRNB: Signed saturating shift right narrow by immediate (bottom).

SQSHRNT: Signed saturating shift right narrow by immediate (top).

SQSHRUNB: Signed saturating shift right unsigned narrow by immediate (bottom).

SQSHRUNT: Signed saturating shift right unsigned narrow by immediate (top).

SQSUB (immediate): Signed saturating subtract immediate (unpredicated).

SQSUB (vectors, predicated): Signed saturating subtraction (predicated).

SQSUB (vectors, unpredicated): Signed saturating subtract vectors (unpredicated).

SQSUBR: Signed saturating subtraction reversed vectors (predicated).

SQXTNB: Signed saturating extract narrow (bottom).

SQXTNT: Signed saturating extract narrow (top).

SQXTUNB: Signed saturating unsigned extract narrow (bottom).

SQXTUNT: Signed saturating unsigned extract narrow (top).

SRHADD: Signed rounding halving addition.

SRI: Shift right and insert (immediate).

SRSHL: Signed rounding shift left by vector (predicated).

SRSHLR: Signed rounding shift left reversed vectors (predicated).

SRSHR: Signed rounding shift right by immediate.

SRSRA: Signed rounding shift right and accumulate (immediate).

SSHLLB: Signed shift left long by immediate (bottom).

SSHLLT: Signed shift left long by immediate (top).

SSRA: Signed shift right and accumulate (immediate).

SSUBLB: Signed subtract long (bottom).

SSUBLBT: Signed subtract long (bottom - top).

SSUBLT: Signed subtract long (top).

SSUBLTB: Signed subtract long (top - bottom).

SSUBWB: Signed subtract wide (bottom).

SSUBWT: Signed subtract wide (top).

ST1B (scalar plus immediate): Contiguous store bytes from vector (immediate index).

ST1B (scalar plus scalar): Contiguous store bytes from vector (scalar index).

ST1B (scalar plus vector): Scatter store bytes from a vector (vector index).

ST1B (vector plus immediate): Scatter store bytes from a vector (immediate index).

ST1D (scalar plus immediate): Contiguous store doublewords from vector (immediate index).

ST1D (scalar plus scalar): Contiguous store doublewords from vector (scalar index).

ST1D (scalar plus vector): Scatter store doublewords from a vector (vector index).

ST1D (vector plus immediate): Scatter store doublewords from a vector (immediate index).

ST1H (scalar plus immediate): Contiguous store halfwords from vector (immediate index).

ST1H (scalar plus scalar): Contiguous store halfwords from vector (scalar index).

ST1H (scalar plus vector): Scatter store halfwords from a vector (vector index).

ST1H (vector plus immediate): Scatter store halfwords from a vector (immediate index).

ST1W (scalar plus immediate): Contiguous store words from vector (immediate index).

ST1W (scalar plus scalar): Contiguous store words from vector (scalar index).

ST1W (scalar plus vector): Scatter store words from a vector (vector index).

ST1W (vector plus immediate): Scatter store words from a vector (immediate index).

ST2B (scalar plus immediate): Contiguous store two-byte structures from two vectors (immediate index).

ST2B (scalar plus scalar): Contiguous store two-byte structures from two vectors (scalar index).

ST2D (scalar plus immediate): Contiguous store two-doubleword structures from two vectors (immediate index).

ST2D (scalar plus scalar): Contiguous store two-doubleword structures from two vectors (scalar index).

ST2H (scalar plus immediate): Contiguous store two-halfword structures from two vectors (immediate index).

ST2H (scalar plus scalar): Contiguous store two-halfword structures from two vectors (scalar index).

ST2W (scalar plus immediate): Contiguous store two-word structures from two vectors (immediate index).

ST2W (scalar plus scalar): Contiguous store two-word structures from two vectors (scalar index).

ST3B (scalar plus immediate): Contiguous store three-byte structures from three vectors (immediate index).

ST3B (scalar plus scalar): Contiguous store three-byte structures from three vectors (scalar index).

ST3D (scalar plus immediate): Contiguous store three-doubleword structures from three vectors (immediate index).

ST3D (scalar plus scalar): Contiguous store three-doubleword structures from three vectors (scalar index).

ST3H (scalar plus immediate): Contiguous store three-halfword structures from three vectors (immediate index).

ST3H (scalar plus scalar): Contiguous store three-halfword structures from three vectors (scalar index).

ST3W (scalar plus immediate): Contiguous store three-word structures from three vectors (immediate index).

ST3W (scalar plus scalar): Contiguous store three-word structures from three vectors (scalar index).

ST4B (scalar plus immediate): Contiguous store four-byte structures from four vectors (immediate index).

ST4B (scalar plus scalar): Contiguous store four-byte structures from four vectors (scalar index).

ST4D (scalar plus immediate): Contiguous store four-doubleword structures from four vectors (immediate index).

ST4D (scalar plus scalar): Contiguous store four-doubleword structures from four vectors (scalar index).

ST4H (scalar plus immediate): Contiguous store four-halfword structures from four vectors (immediate index).

ST4H (scalar plus scalar): Contiguous store four-halfword structures from four vectors (scalar index).

ST4W (scalar plus immediate): Contiguous store four-word structures from four vectors (immediate index).

ST4W (scalar plus scalar): Contiguous store four-word structures from four vectors (scalar index).

STNT1B (scalar plus immediate): Contiguous store non-temporal bytes from vector (immediate index).

STNT1B (scalar plus scalar): Contiguous store non-temporal bytes from vector (scalar index).

STNT1B (vector plus scalar): Scatter store non-temporal bytes.

STNT1D (scalar plus immediate): Contiguous store non-temporal doublewords from vector (immediate index).

STNT1D (scalar plus scalar): Contiguous store non-temporal doublewords from vector (scalar index).

STNT1D (vector plus scalar): Scatter store non-temporal doublewords.

STNT1H (scalar plus immediate): Contiguous store non-temporal halfwords from vector (immediate index).

STNT1H (scalar plus scalar): Contiguous store non-temporal halfwords from vector (scalar index).

STNT1H (vector plus scalar): Scatter store non-temporal halfwords.

STNT1W (scalar plus immediate): Contiguous store non-temporal words from vector (immediate index).

STNT1W (scalar plus scalar): Contiguous store non-temporal words from vector (scalar index).

STNT1W (vector plus scalar): Scatter store non-temporal words.

STR (predicate): Store predicate register.

STR (vector): Store vector register.

SUB (immediate): Subtract immediate (unpredicated).

SUB (vectors, predicated): Subtract vectors (predicated).

SUB (vectors, unpredicated): Subtract vectors (unpredicated).

SUBHNB: Subtract narrow high part (bottom).

SUBHNT: Subtract narrow high part (top).

SUBR (immediate): Reversed subtract from immediate (unpredicated).

SUBR (vectors): Reversed subtract vectors (predicated).

SUDOT: Signed by unsigned integer indexed dot product.

SUNPKHI, SUNPKLO: Signed unpack and extend half of vector.

SUQADD: Signed saturating addition of unsigned value.

SXTB, SXTH, SXTW: Signed byte / halfword / word extend (predicated).

TBL: Programmable table lookup in one or two vector table (zeroing).

TBX: Programmable table lookup in single vector table (merging).

TRN1, TRN2 (predicates): Interleave even or odd elements from two predicates.

TRN1, TRN2 (vectors): Interleave even or odd elements from two vectors.

UABA: Unsigned absolute difference and accumulate.

UABALB: Unsigned absolute difference and accumulate long (bottom).

UABALT: Unsigned absolute difference and accumulate long (top).

UABD: Unsigned absolute difference (predicated).

UABDLB: Unsigned absolute difference long (bottom).

UABDLT: Unsigned absolute difference long (top).

UADALP: Unsigned add and accumulate long pairwise.

UADDLB: Unsigned add long (bottom).

UADDLT: Unsigned add long (top).

UADDV: Unsigned add reduction to scalar.

UADDWB: Unsigned add wide (bottom).

UADDWT: Unsigned add wide (top).

UCLAMP: Unsigned clamp to minimum/maximum vector.

UCVTF: Unsigned integer convert to floating-point (predicated).

UDIV: Unsigned divide (predicated).

UDIVR: Unsigned reversed divide (predicated).

UDOT (indexed): Unsigned integer indexed dot product.

UDOT (vectors): Unsigned integer dot product.

UHADD: Unsigned halving addition.

UHSUB: Unsigned halving subtract.

UHSUBR: Unsigned halving subtract reversed vectors.

UMAX (immediate): Unsigned maximum with immediate (unpredicated).

UMAX (vectors): Unsigned maximum vectors (predicated).

UMAXP: Unsigned maximum pairwise.

UMAXV: Unsigned maximum reduction to scalar.

UMIN (immediate): Unsigned minimum with immediate (unpredicated).

UMIN (vectors): Unsigned minimum vectors (predicated).

UMINP: Unsigned minimum pairwise.

UMINV: Unsigned minimum reduction to scalar.

UMLALB (indexed): Unsigned multiply-add long to accumulator (bottom, indexed).

UMLALB (vectors): Unsigned multiply-add long to accumulator (bottom).

UMLALT (indexed): Unsigned multiply-add long to accumulator (top, indexed).

UMLALT (vectors): Unsigned multiply-add long to accumulator (top).

UMLSLB (indexed): Unsigned multiply-subtract long from accumulator (bottom, indexed).

UMLSLB (vectors): Unsigned multiply-subtract long from accumulator (bottom).

UMLSLT (indexed): Unsigned multiply-subtract long from accumulator (top, indexed).

UMLSLT (vectors): Unsigned multiply-subtract long from accumulator (top).

UMMLA: Unsigned integer matrix multiply-accumulate.

UMULH (predicated): Unsigned multiply returning high half (predicated).

UMULH (unpredicated): Unsigned multiply returning high half (unpredicated).

UMULLB (indexed): Unsigned multiply long (bottom, indexed).

UMULLB (vectors): Unsigned multiply long (bottom).

UMULLT (indexed): Unsigned multiply long (top, indexed).

UMULLT (vectors): Unsigned multiply long (top).

UQADD (immediate): Unsigned saturating add immediate (unpredicated).

UQADD (vectors, predicated): Unsigned saturating addition (predicated).

UQADD (vectors, unpredicated): Unsigned saturating add vectors (unpredicated).

UQDECB: Unsigned saturating decrement scalar by multiple of 8-bit predicate constraint element count.

UQDECD (scalar): Unsigned saturating decrement scalar by multiple of 64-bit predicate constraint element count.

UQDECD (vector): Unsigned saturating decrement vector by multiple of 64-bit predicate constraint element count.

UQDECH (scalar): Unsigned saturating decrement scalar by multiple of 16-bit predicate constraint element count.

UQDECH (vector): Unsigned saturating decrement vector by multiple of 16-bit predicate constraint element count.

UQDECP (scalar): Unsigned saturating decrement scalar by count of true predicate elements.

UQDECP (vector): Unsigned saturating decrement vector by count of true predicate elements.

UQDECW (scalar): Unsigned saturating decrement scalar by multiple of 32-bit predicate constraint element count.

UQDECW (vector): Unsigned saturating decrement vector by multiple of 32-bit predicate constraint element count.

UQINCB: Unsigned saturating increment scalar by multiple of 8-bit predicate constraint element count.

UQINCD (scalar): Unsigned saturating increment scalar by multiple of 64-bit predicate constraint element count.

UQINCD (vector): Unsigned saturating increment vector by multiple of 64-bit predicate constraint element count.

UQINCH (scalar): Unsigned saturating increment scalar by multiple of 16-bit predicate constraint element count.

UQINCH (vector): Unsigned saturating increment vector by multiple of 16-bit predicate constraint element count.

UQINCP (scalar): Unsigned saturating increment scalar by count of true predicate elements.

UQINCP (vector): Unsigned saturating increment vector by count of true predicate elements.

UQINCW (scalar): Unsigned saturating increment scalar by multiple of 32-bit predicate constraint element count.

UQINCW (vector): Unsigned saturating increment vector by multiple of 32-bit predicate constraint element count.

UQRSHL: Unsigned saturating rounding shift left by vector (predicated).

UQRSHLR: Unsigned saturating rounding shift left reversed vectors (predicated).

UQRSHRNB: Unsigned saturating rounding shift right narrow by immediate (bottom).

UQRSHRNT: Unsigned saturating rounding shift right narrow by immediate (top).

UQSHL (immediate): Unsigned saturating shift left by immediate.

UQSHL (vectors): Unsigned saturating shift left by vector (predicated).

UQSHLR: Unsigned saturating shift left reversed vectors (predicated).

UQSHRNB: Unsigned saturating shift right narrow by immediate (bottom).

UQSHRNT: Unsigned saturating shift right narrow by immediate (top).

UQSUB (immediate): Unsigned saturating subtract immediate (unpredicated).

UQSUB (vectors, predicated): Unsigned saturating subtraction (predicated).

UQSUB (vectors, unpredicated): Unsigned saturating subtract vectors (unpredicated).

UQSUBR: Unsigned saturating subtraction reversed vectors (predicated).

UQXTNB: Unsigned saturating extract narrow (bottom).

UQXTNT: Unsigned saturating extract narrow (top).

URECPE: Unsigned reciprocal estimate (predicated).

URHADD: Unsigned rounding halving addition.

URSHL: Unsigned rounding shift left by vector (predicated).

URSHLR: Unsigned rounding shift left reversed vectors (predicated).

URSHR: Unsigned rounding shift right by immediate.

URSQRTE: Unsigned reciprocal square root estimate (predicated).

URSRA: Unsigned rounding shift right and accumulate (immediate).

USDOT (indexed): Unsigned by signed integer indexed dot product.

USDOT (vectors): Unsigned by signed integer dot product.

USHLLB: Unsigned shift left long by immediate (bottom).

USHLLT: Unsigned shift left long by immediate (top).

USMMLA: Unsigned by signed integer matrix multiply-accumulate.

USQADD: Unsigned saturating addition of signed value.

USRA: Unsigned shift right and accumulate (immediate).

USUBLB: Unsigned subtract long (bottom).

USUBLT: Unsigned subtract long (top).

USUBWB: Unsigned subtract wide (bottom).

USUBWT: Unsigned subtract wide (top).

UUNPKHI, UUNPKLO: Unsigned unpack and extend half of vector.

UXTB, UXTH, UXTW: Unsigned byte / halfword / word extend (predicated).

UZP1, UZP2 (predicates): Concatenate even or odd elements from two predicates.

UZP1, UZP2 (vectors): Concatenate even or odd elements from two vectors.

WHILEGE: While decrementing signed scalar greater than or equal to scalar.

WHILEGT: While decrementing signed scalar greater than scalar.

WHILEHI: While decrementing unsigned scalar higher than scalar.

WHILEHS: While decrementing unsigned scalar higher or same as scalar.

WHILELE: While incrementing signed scalar less than or equal to scalar.

WHILELO: While incrementing unsigned scalar lower than scalar.

WHILELS: While incrementing unsigned scalar lower or same as scalar.

WHILELT: While incrementing signed scalar less than scalar.

WHILERW: While free of read-after-write conflicts.

WHILEWR: While free of write-after-read/write conflicts.

WRFFR: Write the first-fault register.

XAR: Bitwise exclusive OR and rotate right by immediate.

ZIP1, ZIP2 (predicates): Interleave elements from two half predicates.

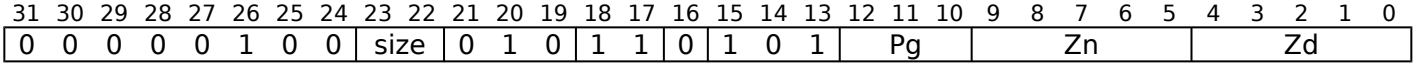
ZIP1, ZIP2 (vectors): Interleave elements from two half vectors.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ABS

Absolute value (predicated)

Compute the absolute value of the signed integer in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.



ABS <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element = SInt(Elem[operand, e, esize]);
        element = Abs(element);
        Elem[result, e, esize] = element<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

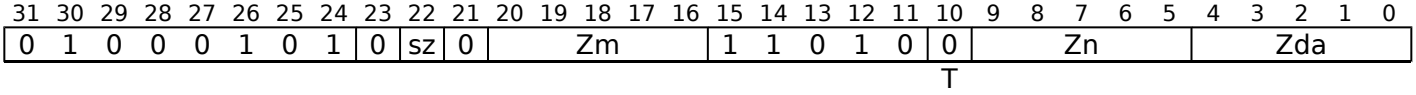
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ADCLB

Add with carry long (bottom)

Add the even-numbered elements of the first source vector and the 1-bit carry from the least-significant bit of the odd-numbered elements of the second source vector to the even-numbered elements of the destination and accumulator vector. The 1-bit carry output is placed in the corresponding odd-numbered element of the destination vector.



ADCLB <Zda>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer pairs = VL DIV (esize * 2);
bits(VL) operand = Z[n, VL];
bits(VL) carries = Z[m, VL];
bits(VL) result = Z[da, VL];

for p = 0 to pairs-1
    bits(esize) element1 = Elem[result, 2*p + 0, esize];
    bits(esize) element2 = Elem[operand, 2*p + 0, esize];
    bit carry_in = Elem[carries, 2*p + 1, esize]<0>;

    (res, nzcv) = AddWithCarry(element1, element2, carry_in);
    carry_out = nzcv<1>;

    Elem[result, 2*p + 0, esize] = res;
    Elem[result, 2*p + 1, esize] = ZeroExtend(carry_out, esize);

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

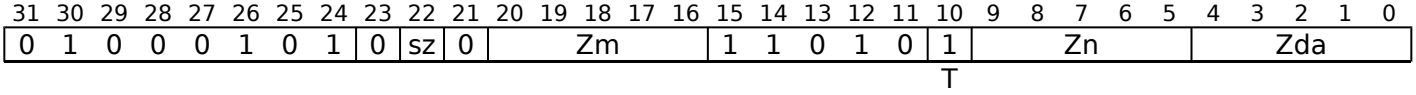
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ADCLT

Add with carry long (top)

Add the odd-numbered elements of the first source vector and the 1-bit carry from the least-significant bit of the odd-numbered elements of the second source vector to the even-numbered elements of the destination and accumulator vector. The 1-bit carry output is placed in the corresponding odd-numbered element of the destination vector.



ADCLT <Zda>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer pairs = VL DIV (esize * 2);
bits(VL) operand = Z[n, VL];
bits(VL) carries = Z[m, VL];
bits(VL) result = Z[da, VL];

for p = 0 to pairs-1
    bits(esize) element1 = Elem[result, 2*p + 0, esize];
    bits(esize) element2 = Elem[operand, 2*p + 1, esize];
    bit carry_in = Elem[carries, 2*p + 1, esize]<0>;

    (res, nzcv) = AddWithCarry(element1, element2, carry_in);
    carry_out = nzcv<1>;

    Elem[result, 2*p + 0, esize] = res;
    Elem[result, 2*p + 1, esize] = ZeroExtend(carry_out, esize);

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

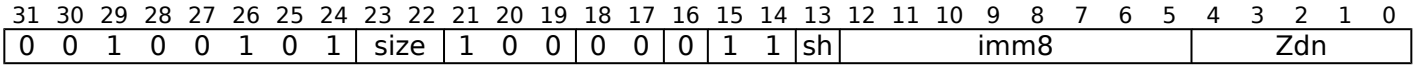
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ADD (immediate)

Add immediate (unpredicated)

Add an unsigned immediate to each element of the source vector, and destructively place the results in the corresponding elements of the source vector. This instruction is unpredicated.
The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.
The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<uimm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".



ADD <Zdn>.<T>, <Zdn>.<T>, #<imm>{, <shift>}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = UInt(imm8);
if sh == '1' then imm = imm << 8;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<imm> Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #8

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    Elem[result, e, esize] = element1 + imm;

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.

    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

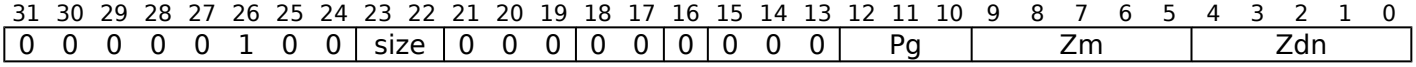
- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ADD (vectors, predicated)

Add vectors (predicated)

Add active elements of the second source vector to corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



ADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = element1 + element2;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

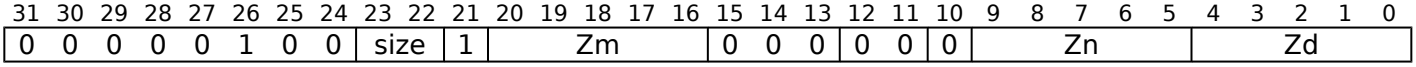
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ADD (vectors, unpredicated)

Add vectors (unpredicated)

Add all elements of the second source vector to corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.



ADD <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = element1 + element2;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

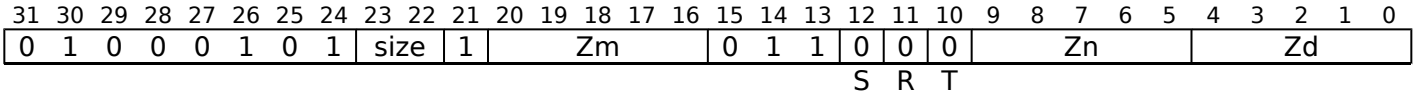
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ADDHNB

Add narrow high part (bottom)

Add each vector element of the first source vector to the corresponding vector element of the second source vector, and place the most significant half of the result in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. This instruction is unpredicated.



ADDHNB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	B
10	H
11	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	H
10	S
11	D

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
constant integer halfesize = esize DIV 2;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer res = (element1 + element2) >> halfesize;
    Elem[result, 2*e + 0, halfesize] = res<halfesize-1:0>;
    Elem[result, 2*e + 1, halfesize] = Zeros(halfesize);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

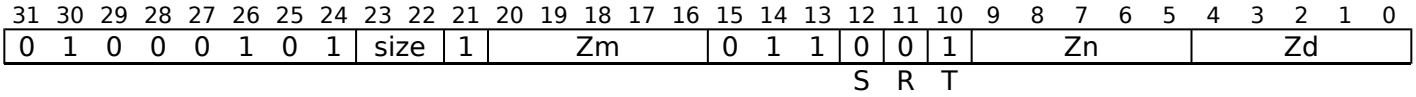
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ADDHNT

Add narrow high part (top)

Add each vector element of the first source vector to the corresponding vector element of the second source vector, and place the most significant half of the result in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. This instruction is unpredicated.



ADDHNT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	B
10	H
11	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	H
10	S
11	D

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[d, VL];
constant integer halfesize = esize DIV 2;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer res = (element1 + element2) >> halfesize;
    Elem[result, 2*e + 1, halfesize] = res<halfesize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

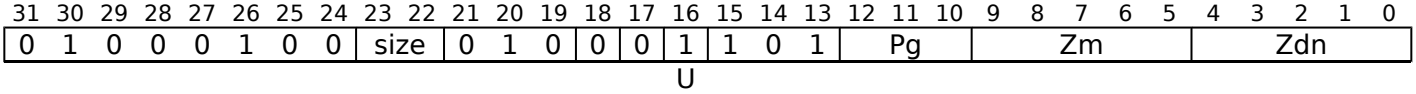
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ADDP

Add pairwise

Add pairs of adjacent elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.



ADDP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
integer element1;
integer element2;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '0' then
        Elem[result, e, esize] = Elem[operand1, e, esize];
    else
        if IsEven(e) then
            element1 = UInt(Elem[operand1, e + 0, esize]);
            element2 = UInt(Elem[operand1, e + 1, esize]);
        else
            element1 = UInt(Elem[operand2, e - 1, esize]);
            element2 = UInt(Elem[operand2, e + 0, esize]);
        integer res = element1 + element2;
        Elem[result, e, esize] = res<esize-1:0>;

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

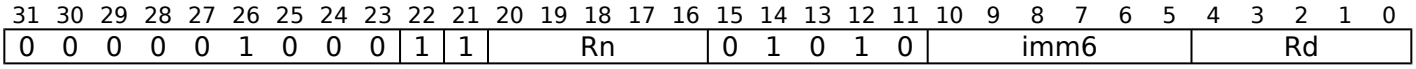
- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ADDPL

Add multiple of predicate register size to scalar register

Add the current predicate register size in bytes multiplied by an immediate in the range -32 to 31 to the 64-bit source general-purpose register or current stack pointer and place the result in the 64-bit destination general-purpose register or current stack pointer.



ADDPL <Xd|SP>, <Xn|SP>, #<imm>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer d = UInt(Rd);
integer imm = SInt(imm6);
```

Assembler Symbols

<Xd|SP>	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xn|SP>	Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<imm>	Is the signed immediate operand, in the range -32 to 31, encoded in the "imm6" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(64) operand1 = if n == 31 then SP[] else X[n, 64];
bits(64) result = operand1 + (imm * (PL DIV 8));

if d == 31 then
    SP[] = result;
else
    X[d, 64] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

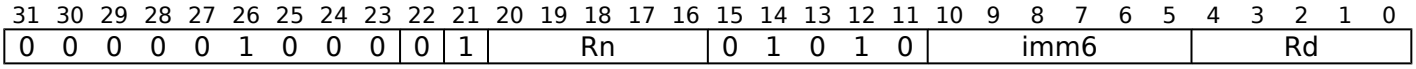
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ADDVL

Add multiple of vector register size to scalar register

Add the current vector register size in bytes multiplied by an immediate in the range -32 to 31 to the 64-bit source general-purpose register or current stack pointer, and place the result in the 64-bit destination general-purpose register or current stack pointer.



ADDVL <Xd|SP>, <Xn|SP>, #<imm>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer d = UInt(Rd);
integer imm = SInt(imm6);
```

Assembler Symbols

<Xd|SP>	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xn|SP>	Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<imm>	Is the signed immediate operand, in the range -32 to 31, encoded in the "imm6" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(64) operand1 = if n == 31 then SP[] else X[n, 64];
bits(64) result = operand1 + (imm * (VL DIV 8));

if d == 31 then
    SP[] = result;
else
    X[d, 64] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

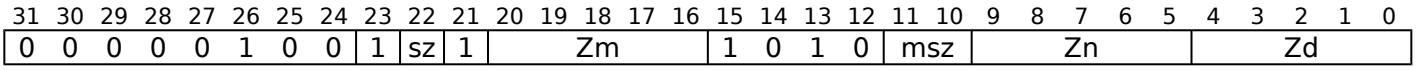
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ADR

Compute vector address

Optionally sign or zero-extend the least significant 32-bits of each element from a vector of offsets or indices in the second source vector, scale each index by 2, 4 or 8, add to a vector of base addresses from the first source vector, and place the resulting addresses in the destination vector. This instruction is unpredicated.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 3 classes: Packed offsets , Unpacked 32-bit signed offsets and Unpacked 32-bit unsigned offsets

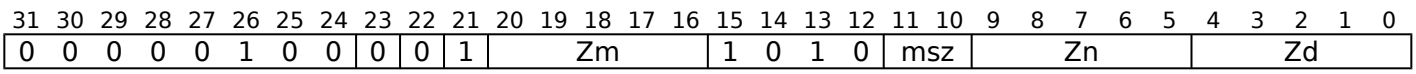
Packed offsets



ADR <Zd>.<T>, [<Zn>.<T>, <Zm>.<T>{, <mod> <amount>}]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
constant integer osize = esize;
boolean unsigned = TRUE;
integer mbytes = 1 << UInt(msz);
```

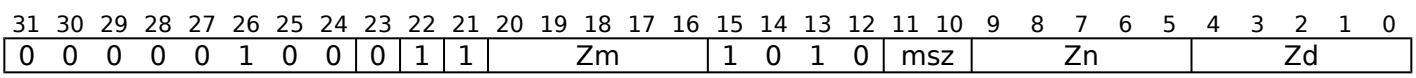
Unpacked 32-bit signed offsets



ADR <Zd>.D, [<Zn>.D, <Zm>.D, SXTW{ <amount>}]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
constant integer osize = 32;
boolean unsigned = FALSE;
integer mbytes = 1 << UInt(msz);
```

Unpacked 32-bit unsigned offsets



ADR <Zd>.D, [<Zn>.D, <Zm>.D, UXTW{ <amount>}]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
constant integer osize = 32;
boolean unsigned = TRUE;
integer mbytes = 1 << UInt(msz);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in “msz”:

msz	<mod>
00	[absent]
x1	LSL
10	LSL

<amount> Is the index shift amount, encoded in “msz”:

msz	<amount>
00	[absent]
01	#1
10	#2
11	#3

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) base = Z[n, VL];
bits(VL) offs = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) addr = Elem[base, e, esize];
    integer offset = Int(Elem[offs, e, esize]<osize-1:0>, unsigned);
    Elem[result, e, esize] = addr + (offset * mbytes);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

AESD

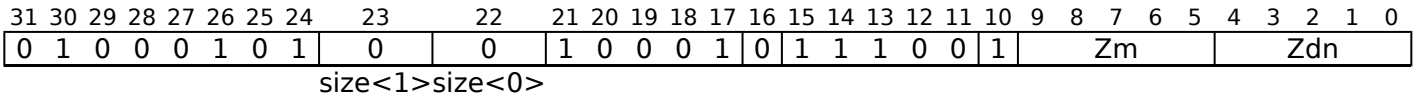
AES single round decryption

The AESD instruction reads a 16-byte state array from each 128-bit segment of the first source vector, together with a round key from the corresponding 128-bit segment of the second source vector. Each state array undergoes a single round of the ADDROUNDKEY(), INVSUBBYTES() and INVSHIFTROWS() transformations in accordance with the AES standard. Each updated state array is destructively placed in the corresponding segment of the first source vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.AES indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE2
(FEAT_SVE_AES)



AESD <Zdn>.B, <Zdn>.B, <Zm>.B

```
if !HaveSVE() || !HaveSVE2AES() then UNDEFINED;
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn>	Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer segments = VL DIV 128;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

result = operand1 EOR operand2;
for s = 0 to segments-1
    Elem[result, s, 128] = AESInvSubBytes(AESInvShiftRows(Elem[result, s, 128]));

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

AESE

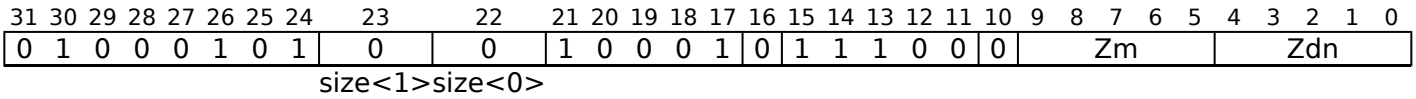
AES single round encryption

The AESE instruction reads a 16-byte state array from each 128-bit segment of the first source vector together with a round key from the corresponding 128-bit segment of the second source vector. Each state array undergoes a single round of the ADDROUNDKEY(), SUBBYTES() and SHIFTROWS() transformations in accordance with the AES standard. Each updated state array is destructively placed in the corresponding segment of the first source vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.AES indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE2
(FEAT_SVE_AES)



AESE <Zdn>.B, <Zdn>.B, <Zm>.B

```
if !HaveSVE() || !HaveSVE2AES() then UNDEFINED;
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn>	Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer segments = VL DIV 128;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

result = operand1 EOR operand2;
for s = 0 to segments-1
    Elem[result, s, 128] = AESSubBytes(AESShiftRows(Elem[result, s, 128]));

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

AESIMC

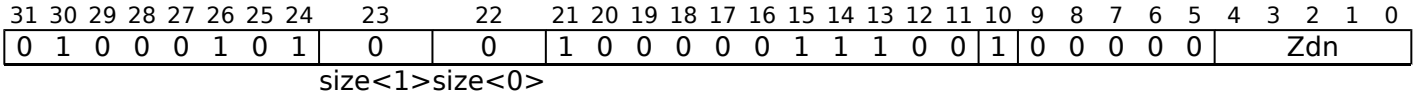
AES inverse mix columns

The AESIMC instruction reads a 16-byte state array from each 128-bit segment of the source register, and performs a single round of the INVMIXCOLUMNS() transformation on each state array in accordance with the AES standard. Each updated state array is destructively placed in the corresponding segment of the first source vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.AES indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE2
(FEAT_SVE_AES)



AESIMC <Zdn>.B, <Zdn>.B

```
if !HaveSVE() || !HaveSVE2AES() then UNDEFINED;
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer segments = VL DIV 128;
bits(VL) operand = Z[dn, VL];
bits(VL) result;

for s = 0 to segments-1
    Elem[result, s, 128] = AESInvMixColumns(Elem[operand, s, 128]);

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

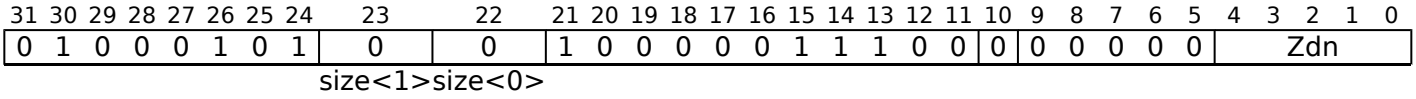
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

AESMC

AES mix columns

The AESMC instruction reads a 16-byte state array from each 128-bit segment of the source register, and performs a single round of the MIXCOLUMNS() transformation on each state array in accordance with the AES standard. Each updated state array is destructively placed in the corresponding segment of the first source vector. This instruction is unpredicated.
ID_AA64ZFR0_EL1.AES indicates whether this instruction is implemented.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE2
(FEAT_SVE_AES)



AESMC <Zdn>.B, <Zdn>.B

```
if !HaveSVE() || !HaveSVE2AES() then UNDEFINED;
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer segments = VL DIV 128;
bits(VL) operand = Z[dn, VL];
bits(VL) result;

for s = 0 to segments-1
    Elem[result, s, 128] = AESMixColumns(Elem[operand, s, 128]);

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

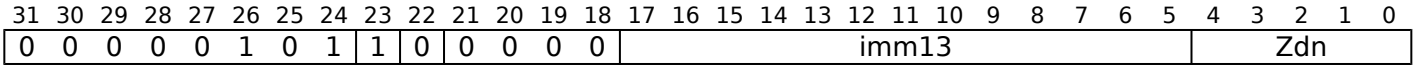
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

AND (immediate)

Bitwise AND with immediate (unpredicated)

Bitwise AND an immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.
This instruction is used by the pseudo-instruction BIC (immediate).



AND <Zdn>.<T>, <Zdn>.<T>, #<const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer dn = UInt(Zdn);
bits(64) imm;
(imm, -) = DecodeBitMasks(imm13<12>, imm13<5:0>, imm13<11:6>, TRUE, 64);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “imm13<12>:imm13<5:0>”:

imm13<12>	imm13<5:0>	<T>
0	0xxxxx	S
0	10xxxx	H
0	110xxx	B
0	1110xx	B
0	11110x	B
0	111110	RESERVED
0	111111	RESERVED
1	xxxxxx	D

<const> Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the "imm13" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 64;
bits(VL) operand = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(64) element1 = Elem[operand, e, 64];
    Elem[result, e, 64] = element1 AND imm;

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

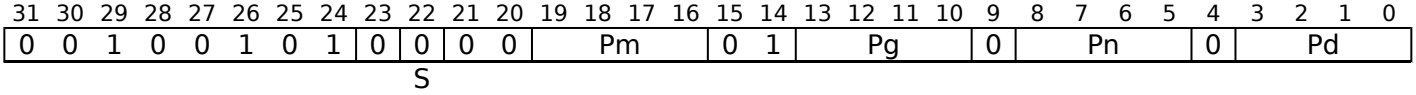
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

AND (predicates)

Bitwise AND predicates

Bitwise AND active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.
This instruction is used by the alias MOV (predicate, predicated, zeroing).



AND <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm>	Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Alias Conditions

Alias	Is preferred when
MOV (predicate, predicated, zeroing)	`S == '0' && Pn == Pm`

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 AND element2;
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

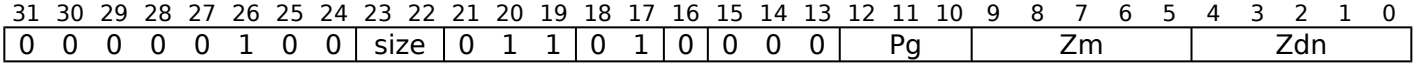
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

AND (vectors, predicated)

Bitwise AND vectors (predicated)

Bitwise AND active elements of the second source vector with corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



AND <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = element1 AND element2;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

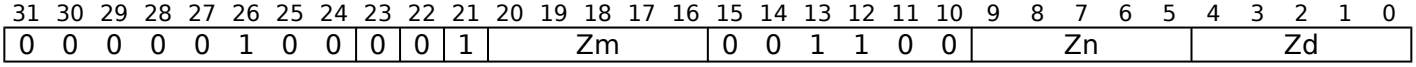
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

AND (vectors, unpredicated)

Bitwise AND vectors (unpredicated)

Bitwise AND all elements of the second source vector with corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.



AND <Zd>.D, <Zn>.D, <Zm>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];

Z[d, VL] = operand1 AND operand2;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

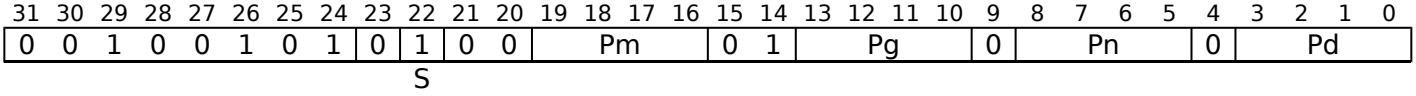
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ANDS

Bitwise AND predicates, setting the condition flags

Bitwise AND active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.
This instruction is used by the alias MOVS (predicated).



ANDS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm>	Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Alias Conditions

Alias	Is preferred when
MOVS (predicated)	`S == '1' && Pn == Pm`

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 AND element2;
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

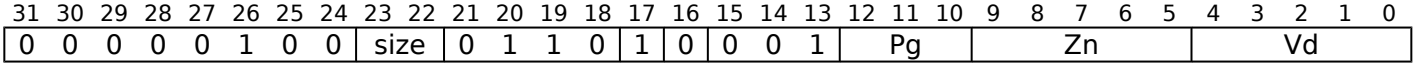
If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ANDV

Bitwise AND reduction to scalar

Bitwise AND horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as all ones.



ANDV <V><d>, <Pg>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(esize) result = Ones(esize);

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        result = result AND Elem[operand, e, esize];

V[d, esize] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

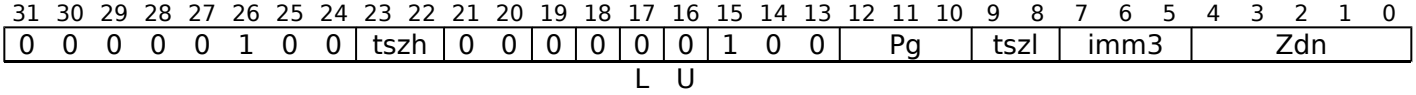
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

ASR (immediate, predicated)

Arithmetic shift right by immediate (predicated)

Shift right by immediate, preserving the sign bit, each active element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. Inactive elements in the destination vector register remain unmodified.



ASR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
case tsize of
    when '0000' UNDEFINED;
    when '0001' esize = 8;
    when '001x' esize = 16;
    when '01xx' esize = 32;
    when '1xxx' esize = 64;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer PL = VL DIV 8;
bits(VL) operand1 = Z[dn, VL];
bits(PL) mask = P[g, PL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = ASR(element1, shift);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

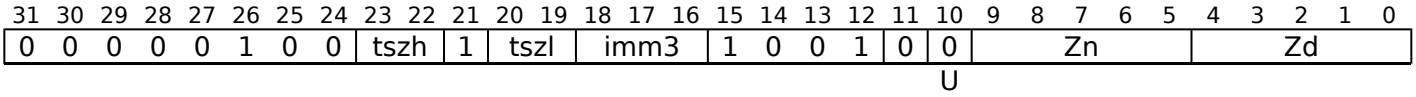
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ASR (immediate, unpredicated)

Arithmetic shift right by immediate (unpredicated)

Shift right by immediate, preserving the sign bit, each element of the source vector, and place the results in the corresponding elements of the destination vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



ASR <Zd>.<T>, <Zn>.<T>, #<const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
case tsize of
    when '0000' UNDEFINED;
    when '0001' esize = 8;
    when '001x' esize = 16;
    when '01xx' esize = 32;
    when '1xxx' esize = 64;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    Elem[result, e, esize] = ASR(element1, shift);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

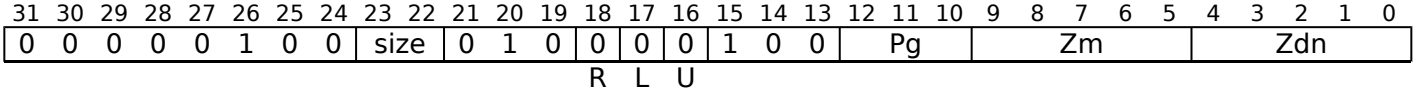
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ASR (vectors)

Arithmetic shift right by vector (predicated)

Shift right, preserving the sign bit, active elements of the first source vector by corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.



ASR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        integer shift = Min(UInt(element2), esize);
        Elem[result, e, esize] = ASR(element1, shift);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

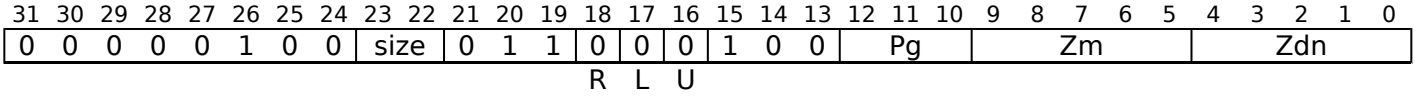
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ASR (wide elements, predicated)

Arithmetic shift right by 64-bit wide elements (predicated)

Shift right, preserving the sign bit, active elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. Inactive elements in the destination vector register remain unmodified.



ASR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	RESERVED

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(64) element2 = Elem[operand2, (e * esize) DIV 64, 64];
        integer shift = Min(UInt(element2), esize);
        Elem[result, e, esize] = ASR(element1, shift);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

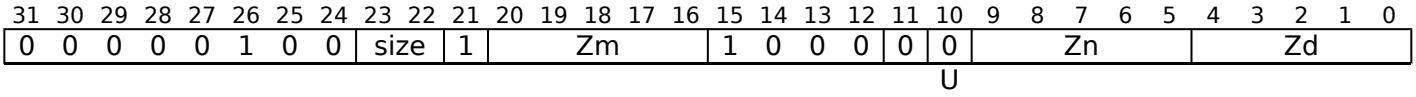
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and destination element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ASR (wide elements, unpredicated)

Arithmetic shift right by 64-bit wide elements (unpredicated)

Shift right, preserving the sign bit, all elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and place the first in the corresponding elements of the destination vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. This instruction is unpredicated.



ASR <Zd>.<T>, <Zn>.<T>, <Zm>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	RESERVED

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(64) element2 = Elem[operand2, (e * esize) DIV 64, 64];
    integer shift = Min(UInt(element2), esize);
    Elem[result, e, esize] = ASR(element1, shift);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

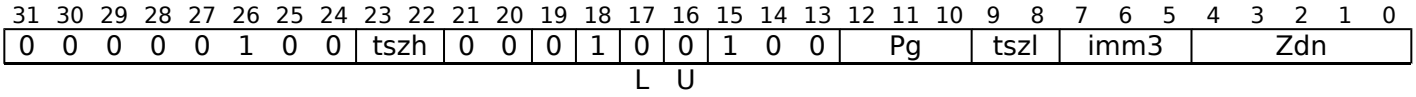
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

- The values of the NZCV flags.

ASRD

Arithmetic shift right for divide by immediate (predicated)

Shift right by immediate, preserving the sign bit, each active element of the source vector, and destructively place the results in the corresponding elements of the source vector. The result rounds toward zero as in a signed division. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. Inactive elements in the destination vector register remain unmodified.



ASRD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
case tsize of
    when '0000' UNDEFINED;
    when '0001' esize = 8;
    when '001x' esize = 16;
    when '01xx' esize = 32;
    when '1xxx' esize = 64;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element1 = SInt(Elem[operand1, e, esize]);
        if element1 < 0 then
            element1 = element1 + ((1 << shift) - 1);
        Elem[result, e, esize] = (element1 >> shift)<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

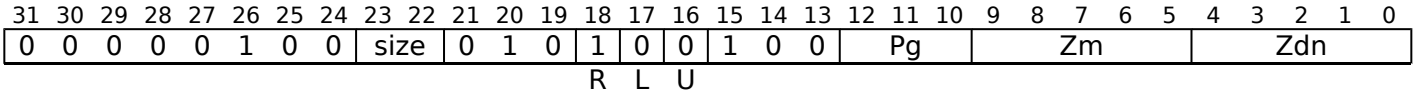
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ASRR

Reversed arithmetic shift right by vector (predicated)

Reversed shift right, preserving the sign bit, active elements of the second source vector by corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.



ASRR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        integer shift = Min(UInt(element1), esize);
        Elem[result, e, esize] = ASR(element2, shift);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

        - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
        - The values of the NZCV flags.
    - The response of this instruction to asynchronous exceptions does not vary based on:
        - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
        - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

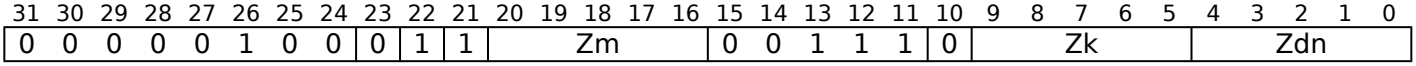
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BCAX

Bitwise clear and exclusive OR

Bitwise AND elements of the second source vector with the corresponding inverted elements of the third source vector, then exclusive OR the results with corresponding elements of the first source vector. The final results are destructively placed in the corresponding elements of the destination and first source vector. This instruction is unpredicated.



BCAX <Zdn>.D, <Zdn>.D, <Zm>.D, <Zk>.D

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
integer m = UInt(Zm);
integer k = UInt(Zk);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn>	Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.
<Zk>	Is the name of the third source scalable vector register, encoded in the "Zk" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[k, VL];

Z[dn, VL] = operand1 EOR (operand2 AND NOT(operand3));
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

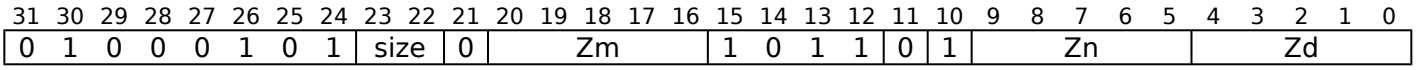
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BDEP

Scatter lower bits into positions selected by bitmask

This instruction scatters the lowest-numbered contiguous bits within each element of the first source vector to the bit positions indicated by non-zero bits in the corresponding mask element of the second source vector, preserving their order, and set the bits corresponding to a zero mask bit to zero. This instruction is unpredicated.
ID_AA64ZFR0_EL1.BitPerm indicates whether this instruction is implemented.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE2
(FEAT_SVE_BitPerm)



BDEP <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() || !HaveSVE2BitPerm() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) data = Z[n, VL];
bits(VL) mask = Z[m, VL];
bits(VL) result;

for e = 0 to elements - 1
    Elem[result, e, esize] = BitDeposit(Elem[data, e, esize], Elem[mask, e, esize]);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

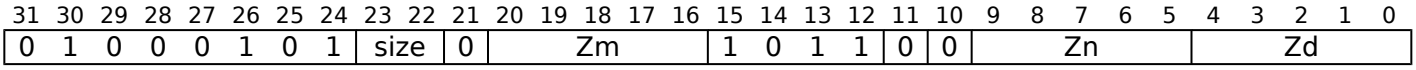
- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

BEXT

Gather lower bits from positions selected by bitmask

This instruction gathers bits in each element of the first source vector from the bit positions indicated by non-zero bits in the corresponding mask element of the second source vector to the lowest-numbered contiguous bits of the corresponding destination element, preserving their order, and sets the remaining higher-numbered bits to zero. This instruction is unpredicated.
ID_AA64ZFR0_EL1.BitPerm indicates whether this instruction is implemented.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE2
(FEAT_SVE_BitPerm)



BEXT <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() || !HaveSVE2BitPerm() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) data = Z[n, VL];
bits(VL) mask = Z[m, VL];
bits(VL) result;

for e = 0 to elements - 1
    Elem[result, e, esize] = BitExtract(Elem[data, e, esize], Elem[mask, e, esize]);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

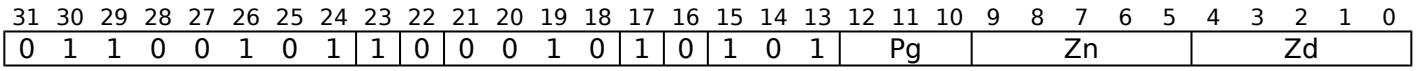
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BFCVT

Floating-point down convert to BFloat16 format (predicated)

Convert to BFloat16 from single-precision in each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.
Since the result type is smaller than the input type, the results are zero-extended to fill each destination element.
ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE
(FEAT_BF16)



BFCVT <Zd>.H, <Pg>/M, <Zn>.S

```
if (!HaveSVE() && !HaveSME()) || !HaveBF16Ext() then UNDEFINED;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, 32) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, 32] == '1' then
        bits(32) element = Elem[operand, e, 32];
        Elem[result, 2*e, 16] = FPConvertBF(element, FPCR[]);
        Elem[result, 2*e+1, 16] = Zeros(16);

Z[d, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

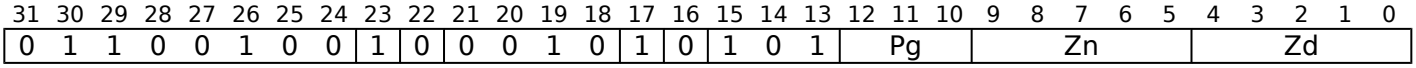
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BFCVTNT

Floating-point down convert and narrow to BFloat16 (top, predicated)

Convert to BFloat16 from single-precision in each active floating-point element of the source vector, and place the results in the odd-numbered 16-bit elements of the destination vector, leaving the even-numbered elements unchanged. Inactive elements in the destination vector register remain unmodified.
ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE
(FEAT_BF16)



BFCVTNT <Zd>.H, <Pg>/M, <Zn>.S

```
if (!HaveSVE() && !HaveSME()) || !HaveBF16Ext() then UNDEFINED;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, 32) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, 32] == '1' then
        bits(32) element = Elem[operand, e, 32];
        Elem[result, 2*e+1, 16] = FPConvertBF(element, FPCR[]);

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BFDOT (indexed)

BFloat16 floating-point indexed dot product

If FEAT_EBF16 is not implemented or FPCR.EBF is 0, irrespective of the other control bits in the FPCR, this instruction:

- Performs an unfused sum-of-products of each pair of adjacent BFloat16 elements in the first source vector with the specified pair of elements in the second vector. The intermediate single-precision products are rounded before they are summed, and the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the first source vector.
- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
- Flushes denormalized inputs and results to zero, as if FPCR.{FZ, FIZ} is {1, 1}.
- Generates only the Default NaN, as if FPCR.DN is 1.

If FEAT_EBF16 is implemented and FPCR.EBF is 1, then this instruction:

- Performs a fused sum-of-products of each pair of adjacent BFloat16 elements in the first source vector with the specified pair of elements in the second vector. The intermediate single-precision products are not rounded before they are summed, but the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the first source vector.
- Generates only the default NaN, as if FPCR.DN is 1.
- Follows all other floating-point behaviors that apply to single-precision arithmetic, as controlled by the effective value of the FPCR in the current execution mode, and captured in the FPSR.

The BFloat16 pairs within the second source vector are specified using an immediate index which selects the same BFloat16 pair position within each 128-bit vector segment. The index range is from 0 to 3.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE

(FEAT_BF16)

31	30	29	28	27	26	25	24	23	22	21	20 19	18 17 16	15	14	13	12	11	10	9 8 7 6 5	4 3 2 1 0
0	1	1	0	0	1	0	0	0	1	1	i2	Zm	0	1	0	0	0	0	Zn	Zda

BFDOT <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if (!HaveSVE() && !HaveSME()) || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i2);
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
<imm>	Is the immediate index, in the range 0 to 3, encoded in the "i2" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
constant integer eltspersegment = 128 DIV 32;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
    bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
    bits(16) elt2_a = Elem[operand2, 2 * s + 0, 16];
    bits(16) elt2_b = Elem[operand2, 2 * s + 1, 16];
    bits(32) sum = Elem[operand3, e, 32];

    sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
    Elem[result, e, 32] = sum;

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BFDOT (vectors)

BFloat16 floating-point dot product

If FEAT_EBF16 is not implemented or FPCR.EBF is 0, irrespective of the other control bits in the FPCR, this instruction:

- Performs an unfused sum-of-products of each pair of adjacent BFloat16 elements in the source vectors. The intermediate single-precision products are rounded before they are summed, and the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the source vectors.
- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
- Flushes denormalized inputs and results to zero, as if FPCR.{FZ, FIZ} is {1, 1}.
- Generates only the Default NaN, as if FPCR.DN is 1.

If FEAT_EBF16 is implemented and FPCR.EBF is 1, then this instruction:

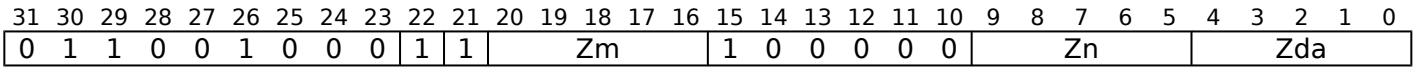
- Performs a fused sum-of-products of each pair of adjacent BFloat16 elements in the source vectors. The intermediate single-precision products are not rounded before they are summed, but the intermediate sum is rounded before accumulation into the single-precision destination element that overlaps with the corresponding pair of BFloat16 elements in the source vectors.
- Generates only the default NaN, as if FPCR.DN is 1.
- Follows all other floating-point behaviors that apply to single-precision arithmetic, as controlled by the effective value of the FPCR in the current execution mode, and captured in the FPSR.

This instruction is unpredicated.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE

(FEAT_BF16)



BFDOT <Zda>.S, <Zn>.H, <Zm>.H

```
if (!HaveSVE() && !HaveSME()) || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
    bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
    bits(16) elt2_a = Elem[operand2, 2 * e + 0, 16];
    bits(16) elt2_b = Elem[operand2, 2 * e + 1, 16];
    bits(32) sum = Elem[operand3, e, 32];

    sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
    Elem[result, e, 32] = sum;

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

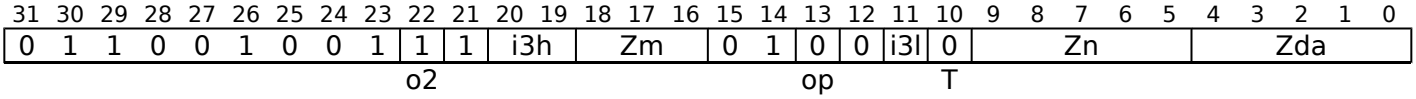
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BFMLALB (indexed)

BFloat16 floating-point multiply-add long to single-precision (bottom, indexed)

This BFloat16 floating-point multiply-add long instruction widens the even-numbered BFloat16 elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the first source vector. This instruction is unpredicated.
ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE
(FEAT_BF16)



BFMLALB <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if (!HaveSVE() && !HaveSME()) || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i3h:i3l);
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
<imm>	Is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
constant integer eltspersegment = 128 DIV 32;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = 2 * segmentbase + index;
    bits(16) element1 = Elem[operand1, 2 * e + 0, 16];
    bits(16) element2 = Elem[operand2, s, 16];
    bits(32) element3 = Elem[operand3, e, 32];
    Elem[result, e, 32] = BFMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

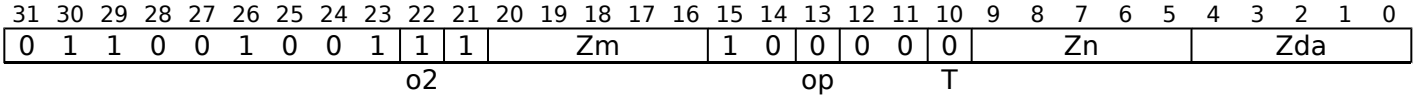
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BFMLALB (vectors)

BFloat16 floating-point multiply-add long to single-precision (bottom)

This BFloat16 floating-point multiply-add long instruction widens the even-numbered BFloat16 elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the source vectors. This instruction is unpredicated.
ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE
(FEAT_BF16)



BFMLALB <Zda>.S, <Zn>.H, <Zm>.H

```
if (!HaveSVE() && !HaveSME()) || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(16) element1 = Elem[operand1, 2 * e + 0, 16];
    bits(16) element2 = Elem[operand2, 2 * e + 0, 16];
    bits(32) element3 = Elem[operand3, e, 32];
    Elem[result, e, 32] = BFMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

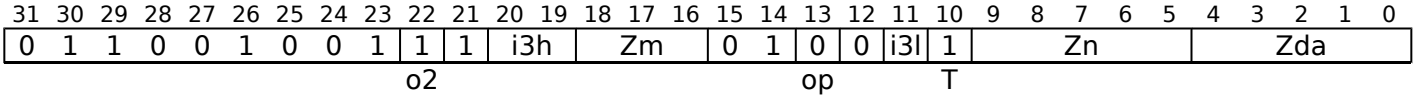
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BFMLALT (indexed)

BFloat16 floating-point multiply-add long to single-precision (top, indexed)

This BFloat16 floating-point multiply-add long instruction widens the odd-numbered BFloat16 elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the first source vector. This instruction is unpredicated.
ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE
(FEAT_BF16)



BFMLALT <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if (!HaveSVE() && !HaveSME()) || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i3h:i3l);
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
<imm>	Is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
constant integer eltspersegment = 128 DIV 32;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = 2 * segmentbase + index;
    bits(16) element1 = Elem[operand1, 2 * e + 1, 16];
    bits(16) element2 = Elem[operand2, s, 16];
    bits(32) element3 = Elem[operand3, e, 32];
    Elem[result, e, 32] = BFMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

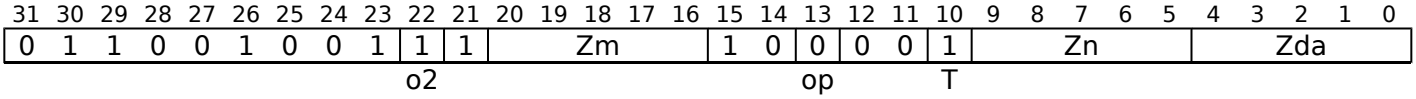
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BFMLALT (vectors)

BFloat16 floating-point multiply-add long to single-precision (top)

This BFloat16 floating-point multiply-add long instruction widens the odd-numbered BFloat16 elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding BFloat16 elements in the source vectors. This instruction is unpredicated.
ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

SVE
(FEAT_BF16)



BFMLALT <Zda>.S, <Zn>.H, <Zm>.H

```
if (!HaveSVE() && !HaveSME()) || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 32;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(16) element1 = Elem[operand1, 2 * e + 1, 16];
    bits(16) element2 = Elem[operand2, 2 * e + 1, 16];
    bits(32) element3 = Elem[operand3, e, 32];
    Elem[result, e, 32] = BFMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BFMMLA

BFloat16 floating-point matrix multiply-accumulate

If FEAT_EBF16 is not implemented or FPCR.EBF is 0, irrespective of the other control bits in the FPCR, this instruction:

- Performs two unfused sums-of-products within each two pairs of adjacent BFloat16 elements while multiplying the 2×4 matrix of BFloat16 values held in each 128-bit segment of the first source vector by the 4×2 matrix of BFloat16 values in the corresponding segment of the second source vector. The intermediate single-precision products are rounded before they are summed and the intermediate sum is rounded before accumulation into the 2×2 single-precision matrix in the corresponding segment of the destination vector. This is equivalent to accumulating two 2-way unfused dot products per destination element.
- Uses the non-IEEE 754 Round-to-Odd rounding mode, which forces bit 0 of an inexact result to 1, and rounds an overflow to an appropriately signed Infinity.
- Does not modify the cumulative FPSR exception bits (IDC, IXC, UFC, OFC, DZC, and IOC).
- Disables trapped floating-point exceptions, as if the FPCR trap enable bits (IDE, IXE, UFE, OFE, DZE, and IOE) are all zero.
- Flushes denormalized inputs and results to zero, as if FPCR.{FZ, FIZ} is {1, 1}.
- Generates only the Default NaN, as if FPCR.DN is 1.

If FEAT_EBF16 is implemented and FPCR.EBF is 1, then this instruction:

- Performs two fused sums-of-products within each two pairs of adjacent BFloat16 elements while multiplying the 2×4 matrix of BFloat16 values held in each 128-bit segment of the first source vector by the 4×2 matrix of BFloat16 values in the corresponding segment of the second source vector. The intermediate single-precision products are not rounded before they are summed, but the intermediate sum is rounded before accumulation into the 2×2 single-precision matrix in the corresponding segment of the destination vector. This is equivalent to accumulating two 2-way fused dot products per destination element.
- Generates only the default NaN, as if FPCR.DN is 1.
- Follows all other floating-point behaviors that apply to single-precision arithmetic, as controlled by the effective value of the FPCR in the current execution mode, and captured in the FPSR.

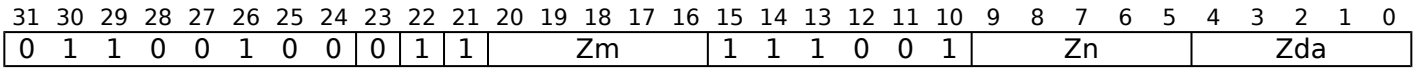
This instruction is unpredicated and vector length agnostic.

ID_AA64ZFR0_EL1.BF16 indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE

(FEAT_BF16)



BFMMLA <Zda>.S, <Zn>.H, <Zm>.H

```
if !HaveSVE() || !HaveBF16Ext() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer segments =  VL DIV 128;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
bits(128) op1, op2;
bits(128) res, addend;

for s = 0 to segments-1
    op1    = Elem[operand1, s, 128];
    op2    = Elem[operand2, s, 128];
    addend = Elem[operand3, s, 128];
    res    = BFMatMulAdd(addend, op1, op2);
    Elem[result, s, 128] = res;

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

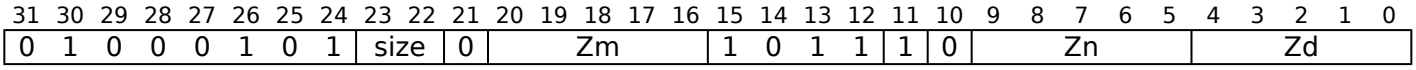
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BGRP

Group bits to right or left as selected by bitmask

This instruction separates bits in each element of the first source vector by gathering from the bit positions indicated by non-zero bits in the corresponding mask element of the second source vector to the lowest-numbered contiguous bits of the corresponding destination element, and from positions indicated by zero bits to the highest-numbered bits of the destination element, preserving the bit order within each group. This instruction is unpredicated.
ID_AA64ZFR0_EL1.BitPerm indicates whether this instruction is implemented.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE2
(FEAT_SVE_BitPerm)



BGRP <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() || !HaveSVE2BitPerm() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) data = Z[n, VL];
bits(VL) mask = Z[m, VL];
bits(VL) result;

for e = 0 to elements - 1
    Elem[result, e, esize] = BitGroup(Elem[data, e, esize], Elem[mask, e, esize]);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BIC (immediate)

Bitwise clear bits using immediate (unpredicated)

Bitwise clear bits using immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This is a pseudo-instruction of AND (immediate). This means:

- The encodings in this description are named to match the encodings of AND (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of AND (immediate) gives the operational pseudocode for this instruction.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	1	0	0	0	0	0	imm13													Zdn				

BIC <Zdn>.<T>, <Zdn>.<T>, #<const>

is equivalent to

AND <Zdn>.<T>, <Zdn>.<T>, #(-<const> - 1)

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “imm13<12>:imm13<5:0>”:

imm13<12>	imm13<5:0>	<T>
0	0xxxxx	S
0	10xxxx	H
0	110xxx	B
0	1110xx	B
0	11110x	B
0	111110	RESERVED
0	111111	RESERVED
1	xxxxxx	D

<const> Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the "imm13" field.

Operation

The description of AND (immediate) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

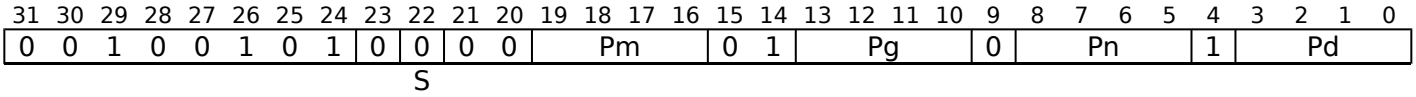
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BIC (predicates)

Bitwise clear predicates

Bitwise AND inverted active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.



BIC <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm>	Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 AND (NOT element2);
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

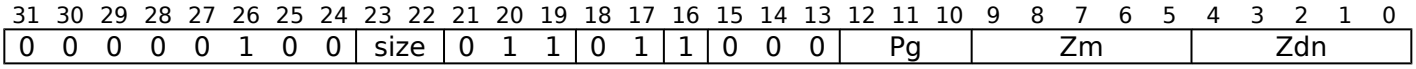
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

- The values of the NZCV flags.

BIC (vectors, predicated)

Bitwise clear vectors (predicated)

Bitwise AND inverted active elements of the second source vector with corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



BIC <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = element1 AND (NOT element2);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

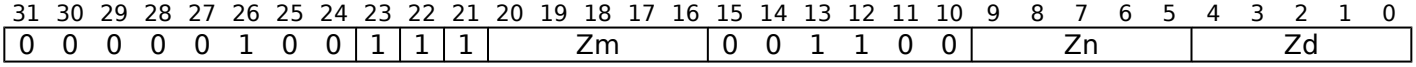
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BIC (vectors, unpredicated)

Bitwise clear vectors (unpredicated)

Bitwise AND inverted all elements of the second source vector with corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.



BIC <Zd>.D, <Zn>.D, <Zm>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];

Z[d, VL] = operand1 AND (NOT operand2);
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

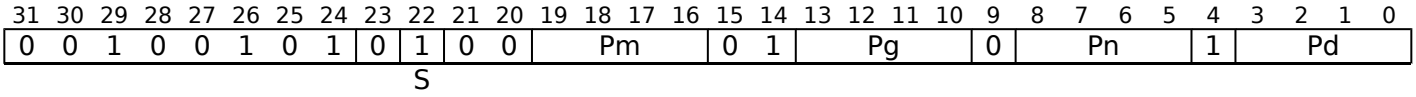
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BICS

Bitwise clear predicates, setting the condition flags

Bitwise AND inverted active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.



BICS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm>	Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 AND (NOT element2);
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

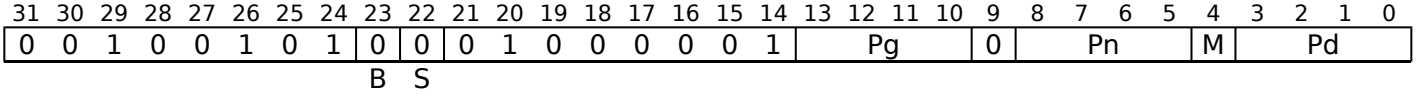
If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BRKA

Break after first true condition

Sets destination predicate elements up to and including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected. Does not set the condition flags.



BRKA <Pd>.B, <Pg>/<ZM>, <Pn>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean merging = (M == '1');
boolean setflags = FALSE;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<ZM> Is the predication qualifier, encoded in “M”:

M	<ZM>
0	Z
1	M

<Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

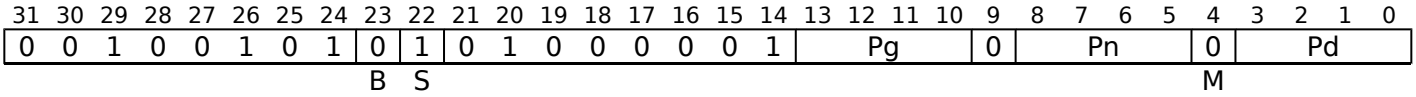
Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand  = P[n, PL];
bits(PL) operand2 = P[d, PL];
boolean break = FALSE;
bits(PL) result;

for e = 0 to elements-1
    boolean element = ElemP[operand, e, esize] == '1';
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = if !break then '1' else '0';
        break = break || element;
    elsif merging then
        ElemP[result, e, esize] = ElemP[operand2, e, esize];
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

BRKAS

Break after first true condition, setting the condition flags

Sets destination predicate elements up to and including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.



BRKAS <Pd>.B, <Pg>/Z, <Pn>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean merging = FALSE;
boolean setflags = TRUE;
```

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the source scalable predicate register, encoded in the "Pn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand  = P[n, PL];
bits(PL) operand2 = P[d, PL];
boolean break = FALSE;
bits(PL) result;

for e = 0 to elements-1
    boolean element = ElemP[operand, e, esize] == '1';
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = if !break then '1' else '0';
        break = break || element;
    elsif merging then
        ElemP[result, e, esize] = ElemP[operand2, e, esize];
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

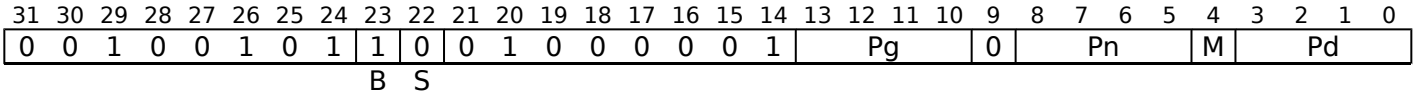
If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BRKB

Break before first true condition

Sets destination predicate elements up to but not including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register remain unmodified or are set to zero, depending on whether merging or zeroing predication is selected. Does not set the condition flags.



BRKB <Pd>.B, <Pg>/<ZM>, <Pn>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean merging = (M == '1');
boolean setflags = FALSE;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<ZM> Is the predication qualifier, encoded in “M”:

M	<ZM>
0	Z
1	M

<Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

Operation

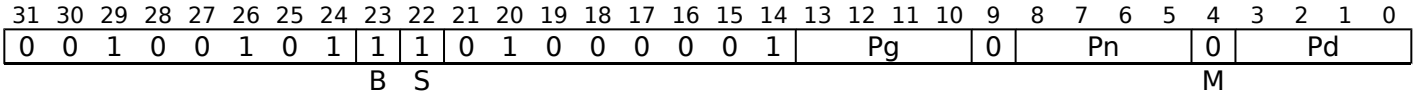
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand  = P[n, PL];
bits(PL) operand2 = P[d, PL];
boolean break = FALSE;
bits(PL) result;

for e = 0 to elements-1
    boolean element = ElemP[operand, e, esize] == '1';
    if ElemP[mask, e, esize] == '1' then
        break = break || element;
        ElemP[result, e, esize] = if !break then '1' else '0';
    elsif merging then
        ElemP[result, e, esize] = ElemP[operand2, e, esize];
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BRKBS

Break before first true condition, setting the condition flags

Sets destination predicate elements up to but not including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.



BRKBS <Pd>.B, <Pg>/Z, <Pn>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean merging = FALSE;
boolean setflags = TRUE;
```

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the source scalable predicate register, encoded in the "Pn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand  = P[n, PL];
bits(PL) operand2 = P[d, PL];
boolean break = FALSE;
bits(PL) result;

for e = 0 to elements-1
    boolean element = ElemP[operand, e, esize] == '1';
    if ElemP[mask, e, esize] == '1' then
        break = break || element;
        ElemP[result, e, esize] = if !break then '1' else '0';
    elsif merging then
        ElemP[result, e, esize] = ElemP[operand2, e, esize];
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

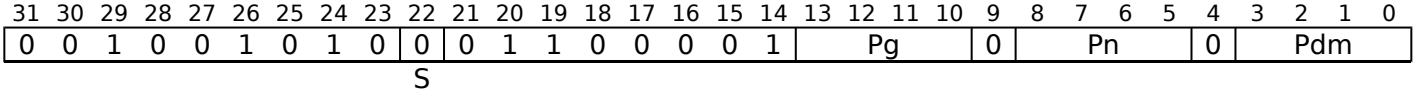
If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BRKN

Propagate break to next partition

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise leaves the destination and second source predicate unchanged. Does not set the condition flags.



BRKN <Pdm>.B, <Pg>/Z, <Pn>.B, <Pdm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer dm = UInt(Pdm);
boolean setflags = FALSE;
```

Assembler Symbols

<Pdm>	Is the name of the second source and destination scalable predicate register, encoded in the "Pdm" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

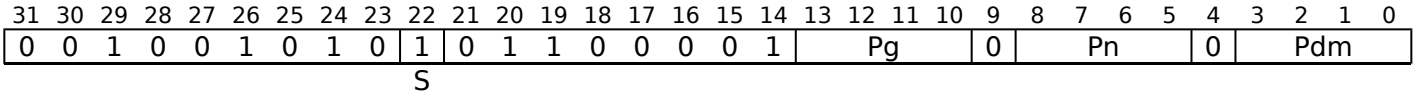
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[dm, PL];
bits(PL) result;

if LastActive(mask, operand1, 8) == '1' then
    result = operand2;
else
    result = Zeros(PL);

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(Ones(PL), result, 8);
P[dm, PL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BRKNS

Propagate break to next partition, setting the condition flags

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise leaves the destination and second source predicate unchanged. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.



BRKNS <Pdm>.B, <Pg>/Z, <Pn>.B, <Pdm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer dm = UInt(Pdm);
boolean setflags = TRUE;
```

Assembler Symbols

<Pdm>	Is the name of the second source and destination scalable predicate register, encoded in the "Pdm" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[dm, PL];
bits(PL) result;

if LastActive(mask, operand1, 8) == '1' then
    result = operand2;
else
    result = Zeros(PL);

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(Ones(PL), result, 8);
P[dm, PL] = result;
```

Operational information

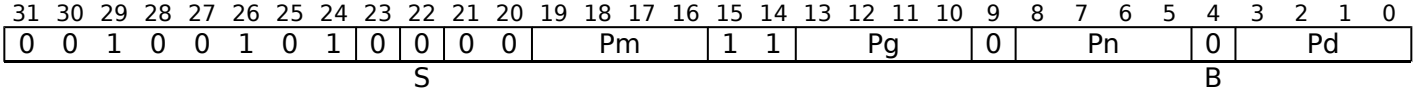
If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BRKPA

Break after first true condition, propagating from previous partition

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise sets destination predicate elements up to and including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.



BRKPA <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm>	Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

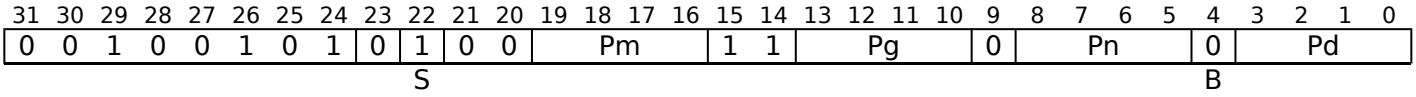
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
boolean last = (LastActive(mask, operand1, 8) == '1');

for e = 0 to elements-1
    if ElemP[mask, e, 8] == '1' then
        ElemP[result, e, 8] = if last then '1' else '0';
        last = last && (ElemP[operand2, e, 8] == '0');
    else
        ElemP[result, e, 8] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BRKPAS

Break after first true condition, propagating from previous partition and setting the condition flags

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise sets destination predicate elements up to and including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.



BRKPAS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm>	Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
boolean last = (LastActive(mask, operand1, 8) == '1');

for e = 0 to elements-1
    if ElemP[mask, e, 8] == '1' then
        ElemP[result, e, 8] = if last then '1' else '0';
        last = last && (ElemP[operand2, e, 8] == '0');
    else
        ElemP[result, e, 8] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

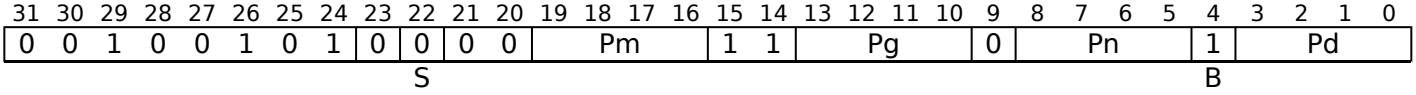
If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BRKPB

Break before first true condition, propagating from previous partition

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise sets destination predicate elements up to but not including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.



BRKPB <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm>	Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

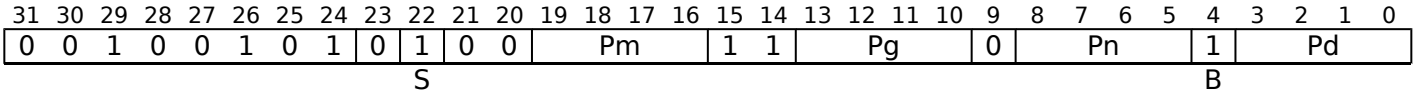
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
boolean last = (LastActive(mask, operand1, 8) == '1');

for e = 0 to elements-1
    if ElemP[mask, e, 8] == '1' then
        last = last && (ElemP[operand2, e, 8] == '0');
        ElemP[result, e, 8] = if last then '1' else '0';
    else
        ElemP[result, e, 8] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BRKPBS

Break before first true condition, propagating from previous partition and setting the condition flags

If the last active element of the first source predicate is false then set the destination predicate to all-false. Otherwise sets destination predicate elements up to but not including the first active and true source element to true, then sets subsequent elements to false. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.



BRKPBS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm>	Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;
boolean last = (LastActive(mask, operand1, 8) == '1');

for e = 0 to elements-1
    if ElemP[mask, e, 8] == '1' then
        last = last && (ElemP[operand2, e, 8] == '0');
        ElemP[result, e, 8] = if last then '1' else '0';
    else
        ElemP[result, e, 8] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

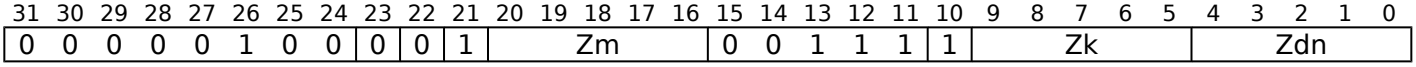
If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BSL

Bitwise select

Selects bits from the first source vector where the corresponding bit in the third source vector is '1', and from the second source vector where the corresponding bit in the third source vector is '0'. The result is placed destructively in the destination and first source vector. This instruction is unpredicated.



BSL <Zdn>.D, <Zdn>.D, <Zm>.D, <Zk>.D

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
integer m = UInt(Zm);
integer k = UInt(Zk);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn>	Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.
<Zk>	Is the name of the third source scalable vector register, encoded in the "Zk" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[k, VL];

Z[dn, VL] = (operand1 AND operand3) OR (operand2 AND NOT(operand3));
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

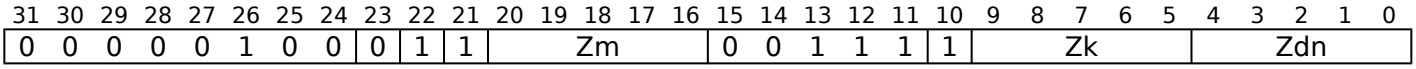
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BSL1N

Bitwise select with first input inverted

Selects bits from the inverted first source vector where the corresponding bit in the third source vector is '1', and from the second source vector where the corresponding bit in the third source vector is '0'. The result is placed destructively in the destination and first source vector. This instruction is unpredicated.



BSL1N <Zdn>.D, <Zdn>.D, <Zm>.D, <Zk>.D

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
integer m = UInt(Zm);
integer k = UInt(Zk);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn>	Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.
<Zk>	Is the name of the third source scalable vector register, encoded in the "Zk" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[k, VL];

Z[dn, VL] = (NOT(operand1) AND operand3) OR (operand2 AND NOT(operand3));
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

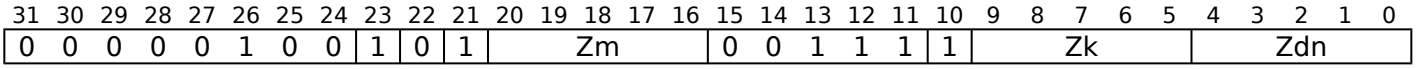
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BSL2N

Bitwise select with second input inverted

Selects bits from the first source vector where the corresponding bit in the third source vector is '1', and from the inverted second source vector where the corresponding bit in the third source vector is '0'. The result is placed destructively in the destination and first source vector. This instruction is unpredicated.



BSL2N <Zdn>.D, <Zdn>.D, <Zm>.D, <Zk>.D

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
integer m = UInt(Zm);
integer k = UInt(Zk);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn>	Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.
<Zk>	Is the name of the third source scalable vector register, encoded in the "Zk" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[k, VL];

Z[dn, VL] = (operand1 AND operand3) OR (NOT(operand2) AND NOT(operand3));
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

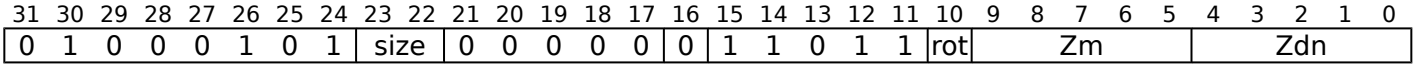
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CADD

Complex integer add with rotate

Add the real and imaginary components of the integral complex numbers from the first source vector to the complex numbers from the second source vector which have first been rotated by 90 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation, equivalent to multiplying the complex numbers in the second source vector by ±J beforehand. Destructively place the results in the corresponding elements of the first source vector. This instruction is unpredicated.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.



CADD <Zdn>.<T>, <Zdn>.<T>, <Zm>.<T>, <const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
boolean sub_i = (rot == '0');
boolean sub_r = (rot == '1');
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<const> Is the const specifier, encoded in “rot”:

rot	<const>
0	#90
1	#270

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (2 * esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for p = 0 to pairs-1
    integer acc_r  = SInt(Elem[operand1, 2 * p + 0, esize]);
    integer acc_i  = SInt(Elem[operand1, 2 * p + 1, esize]);
    integer elt2_r = SInt(Elem[operand2, 2 * p + 0, esize]);
    integer elt2_i = SInt(Elem[operand2, 2 * p + 1, esize]);
    if sub_i then
        acc_r = acc_r - elt2_i;
        acc_i = acc_i + elt2_r;
    if sub_r then
        acc_r = acc_r + elt2_i;
        acc_i = acc_i - elt2_r;

    Elem[result, 2 * p + 0, esize] = acc_r<esize-1:0>;
    Elem[result, 2 * p + 1, esize] = acc_i<esize-1:0>;

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CDOT (indexed)

Complex integer dot product (indexed)

The complex integer dot product instructions delimit the source vectors into pairs of 8-bit or 16-bit signed integer complex numbers. Within each pair, the complex numbers in the first source vector are multiplied by the corresponding complex numbers in the second source vector and the resulting wide real or wide imaginary part of the product is accumulated into a 32-bit or 64-bit destination vector element which overlaps all four of the elements that comprise a pair of complex number values in the first source vector.

As a result each instruction implicitly deinterleaves the real and imaginary components of their complex number inputs, so that the destination vector accumulates 4×wide real sums or 4×wide imaginary sums.

The complex numbers in the second source vector are rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation, by performing the following transformations prior to the dot product operations:

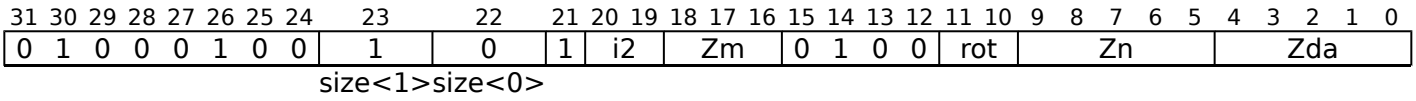
- If the rotation is #0, the imaginary parts of the complex numbers in the second source vector are negated. The destination vector therefore accumulates the real parts of a complex dot product.
- If the rotation is #90, the real and imaginary parts of the complex numbers the second source vector are swapped. The destination vector therefore accumulates the imaginary parts of a complex dot product.
- If the rotation is #180, there is no transformation. The destination vector therefore accumulates the real parts of a complex conjugate dot product.
- If the rotation is #270, the real parts of the complex numbers in the second source vector are negated and then swapped with the imaginary parts. The destination vector therefore accumulates the imaginary parts of a complex conjugate dot product.

The indexed form of these instructions select a single pair of complex numbers within each 128-bit segment of the second source vector as the multiplier for all pairs of complex numbers within the corresponding 128-bit segment of the first source vector. The complex number pairs within the second source vector are specified using an immediate index which selects the same complex number pair position within each 128-bit vector segment. The index range is from 0 to one less than the number of complex number pairs per 128-bit segment, encoded in 1 or 2 bits depending on the size of the complex number pair.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

It has encodings from 2 classes: 32-bit and 64-bit

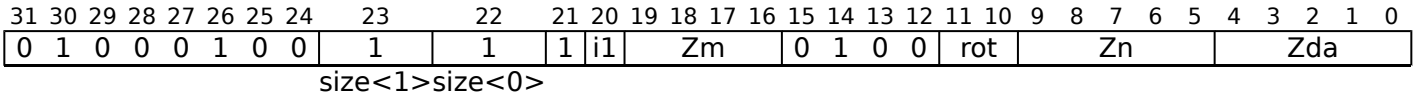
32-bit



CDOT <Zda>.S, <Zn>.B, <Zm>.B[<imm>], <const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_i = (rot<0> == rot<1>);
```

64-bit



CDOT <Zda>.D, <Zn>.H, <Zm>.H[<imm>], <const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_i = (rot<0> == rot<1>);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the 32-bit variant: is the immediate index of a 32-bit group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.

For the 64-bit variant: is the immediate index of a 64-bit group of four 16-bit values within each 128-bit vector segment, in the range 0 to 1, encoded in the "i1" field.

<const> Is the const specifier, encoded in “rot”:

rot	<const>
00	#0
01	#90
10	#180
11	#270

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 1
        integer elt1_r = SInt(Elem[operand1, 4 * e + 2 * i + 0, esize DIV 4]);
        integer elt1_i = SInt(Elem[operand1, 4 * e + 2 * i + 1, esize DIV 4]);
        integer elt2_a = SInt(Elem[operand2, 4 * s + 2 * i + sel_a, esize DIV 4]);
        integer elt2_b = SInt(Elem[operand2, 4 * s + 2 * i + sel_b, esize DIV 4]);
        if sub_i then
            res = res + (elt1_r * elt2_a) - (elt1_i * elt2_b);
        else
            res = res + (elt1_r * elt2_a) + (elt1_i * elt2_b);
    Elem[result, e, esize] = res;

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CDOT (vectors)

Complex integer dot product

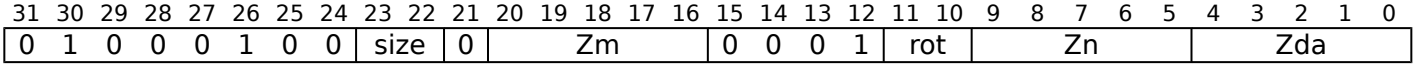
The complex integer dot product instructions delimit the source vectors into pairs of 8-bit or 16-bit signed integer complex numbers. Within each pair, the complex numbers in the first source vector are multiplied by the corresponding complex numbers in the second source vector and the resulting wide real or wide imaginary part of the product is accumulated into a 32-bit or 64-bit destination vector element which overlaps all four of the elements that comprise a pair of complex number values in the first source vector.

As a result each instruction implicitly deinterleaves the real and imaginary components of their complex number inputs, so that the destination vector accumulates 4×wide real sums or 4×wide imaginary sums.

The complex numbers in the second source vector are rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation, by performing the following transformations prior to the dot product operations:

- If the rotation is #0, the imaginary parts of the complex numbers in the second source vector are negated. The destination vector therefore accumulates the real parts of a complex dot product.
- If the rotation is #90, the real and imaginary parts of the complex numbers the second source vector are swapped. The destination vector therefore accumulates the imaginary parts of a complex dot product.
- If the rotation is #180, there is no transformation. The destination vector therefore accumulates the real parts of a complex conjugate dot product.
- If the rotation is #270, the real parts of the complex numbers in the second source vector are negated and then swapped with the imaginary parts. The destination vector therefore accumulates the imaginary parts of a complex conjugate dot product.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.



CDOT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>, <const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_i = (rot<0> == rot<1>);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size<0>”:

size<0>	<T>
0	S
1	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size<0>”:

size<0>	<Tb>
0	B
1	H

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<const> Is the const specifier, encoded in “rot”:

rot	<const>
00	#0
01	#90
10	#180
11	#270

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 1
        integer elt1_r = SInt(Elem[operand1, 4 * e + 2 * i + 0, esize DIV 4]);
        integer elt1_i = SInt(Elem[operand1, 4 * e + 2 * i + 1, esize DIV 4]);
        integer elt2_a = SInt(Elem[operand2, 4 * e + 2 * i + sel_a, esize DIV 4]);
        integer elt2_b = SInt(Elem[operand2, 4 * e + 2 * i + sel_b, esize DIV 4]);
        if sub_i then
            res = res + (elt1_r * elt2_a) - (elt1_i * elt2_b);
        else
            res = res + (elt1_r * elt2_a) + (elt1_i * elt2_b);
    Elem[result, e, esize] = res;

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

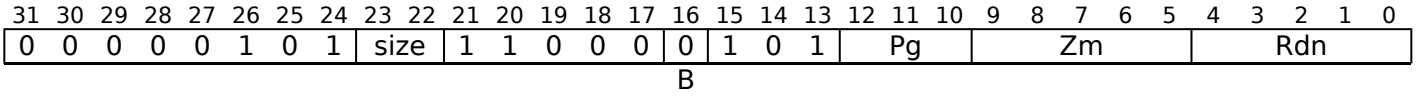
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CLASTA (scalar)

Conditionally extract element after last to general-purpose register

From the source vector register extract the element after the last active element, or if the last active element is the final element extract element zero, and then zero-extend that element to destructively place in the destination and first source general-purpose register. If there are no active elements then destructively zero-extend the least significant element-size bits of the destination and first source general-purpose register.



CLASTA <R><dn>, <Pg>, <R><dn>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Rdn);
integer m = UInt(Zm);
constant integer csize = if esize < 64 then 32 else 64;
boolean isBefore = FALSE;
```

Assembler Symbols

<R> Is a width specifier, encoded in “size”:

size	<R>
01	W
x0	W
11	X

<dn> Is the number [0-30] of the source and destination general-purpose register or the name ZR (31), encoded in the "Rdn" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the source scalable vector register, encoded in the "Zm" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(esize) operand1 = X[dn, esize];
bits(VL) operand2 = Z[m, VL];
bits(csize) result;
integer last = LastActiveElement(mask, esize);

if last < 0 then
    result = ZeroExtend(operand1, csize);
else
    if !isBefore then
        last = last + 1;
        if last >= elements then last = 0;
    result = ZeroExtend(Elem[operand2, last, esize], csize);

X[dn, csize] = result;
```

Operational information

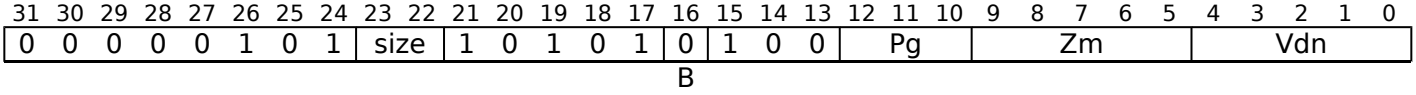
If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CLASTA (SIMD&FP scalar)

Conditionally extract element after last to SIMD&FP scalar register

From the source vector register extract the element after the last active element, or if the last active element is the final element extract element zero, and then zero-extend that element to destructively place in the destination and first source SIMD & floating-point scalar register. If there are no active elements then destructively zero-extend the least significant element-size bits of the destination and first source SIMD & floating-point scalar register.



CLASTA <V><dn>, <Pg>, <V><dn>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Vdn);
integer m = UInt(Zm);
boolean isBefore = FALSE;
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<dn> Is the number [0-31] of the source and destination SIMD&FP register, encoded in the "Vdn" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the source scalable vector register, encoded in the "Zm" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

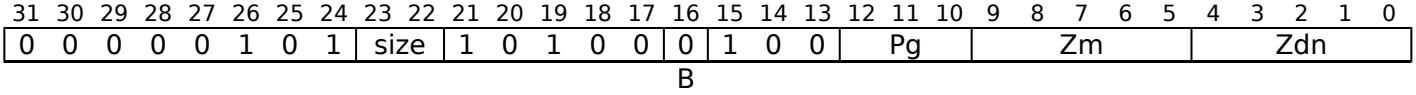
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(esize) operand1 = V[dn, esize];
bits(VL) operand2 = Z[m, VL];
bits(esize) result;
integer last = LastActiveElement(mask, esize);

if last < 0 then
    result = ZeroExtend(operand1, esize);
else
    if !isBefore then
        last = last + 1;
        if last >= elements then last = 0;
    result = Elem[operand2, last, esize];

V[dn, esize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CLASTA (vectors)

Conditionally extract element after last to vector register

From the second source vector register extract the element after the last active element, or if the last active element is the final element extract element zero, and then replicate that element to destructively fill the destination and first source vector.
If there are no active elements then leave the destination and source vector unmodified.



CLASTA <Zdn>.<T>, <Pg>, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean isBefore = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
integer last = LastActiveElement(mask, esize);

if last < 0 then
    result = operand1;
else
    if !isBefore then
        last = last + 1;
        if last >= elements then last = 0;
    for e = 0 to elements-1
        Elem[result, e, esize] = Elem[operand2, last, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

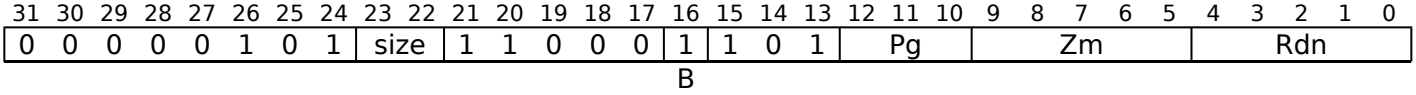
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CLASTB (scalar)

Conditionally extract last element to general-purpose register

From the source vector register extract the last active element, and then zero-extend that element to destructively place in the destination and first source general-purpose register. If there are no active elements then destructively zero-extend the least significant element-size bits of the destination and first source general-purpose register.



CLASTB <R><dn>, <Pg>, <R><dn>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Rdn);
integer m = UInt(Zm);
constant integer csize = if esize < 64 then 32 else 64;
boolean isBefore = TRUE;
```

Assembler Symbols

<R> Is a width specifier, encoded in “size”:

size	<R>
01	W
x0	W
11	X

<dn> Is the number [0-30] of the source and destination general-purpose register or the name ZR (31), encoded in the "Rdn" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the source scalable vector register, encoded in the "Zm" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(esize) operand1 = X[dn, esize];
bits(VL) operand2 = Z[m, VL];
bits(csize) result;
integer last = LastActiveElement(mask, esize);

if last < 0 then
    result = ZeroExtend(operand1, csize);
else
    if !isBefore then
        last = last + 1;
        if last >= elements then last = 0;
    result = ZeroExtend(Elem[operand2, last, esize], csize);

X[dn, csize] = result;
```

Operational information

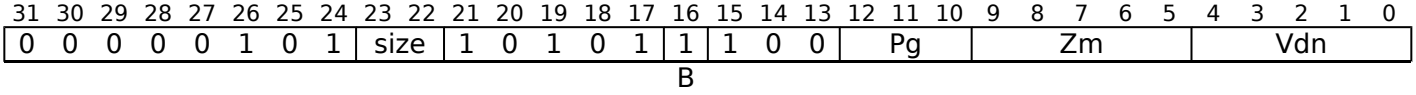
If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CLASTB (SIMD&FP scalar)

Conditionally extract last element to SIMD&FP scalar register

From the source vector register extract the last active element, and then zero-extend that element to destructively place in the destination and first source SIMD & floating-point scalar register. If there are no active elements then destructively zero-extend the least significant element-size bits of the destination and first source SIMD & floating-point scalar register.



CLASTB <V><dn>, <Pg>, <V><dn>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Vdn);
integer m = UInt(Zm);
boolean isBefore = TRUE;
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<dn> Is the number [0-31] of the source and destination SIMD&FP register, encoded in the "Vdn" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the source scalable vector register, encoded in the "Zm" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

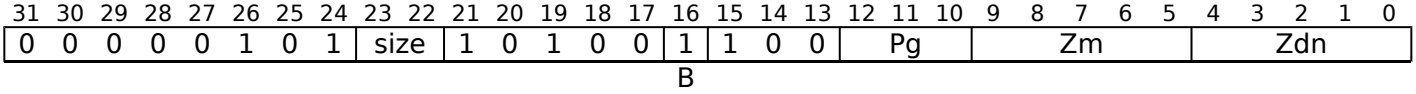
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(esize) operand1 = V[dn, esize];
bits(VL) operand2 = Z[m, VL];
bits(esize) result;
integer last = LastActiveElement(mask, esize);

if last < 0 then
    result = ZeroExtend(operand1, esize);
else
    if !isBefore then
        last = last + 1;
        if last >= elements then last = 0;
    result = Elem[operand2, last, esize];

V[dn, esize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CLASTB (vectors)

Conditionally extract last element to vector register

From the second source vector register extract the last active element, and then replicate that element to destructively fill the destination and first source vector.
If there are no active elements then leave the destination and source vector unmodified.



CLASTB <Zdn>.<T>, <Pg>, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean isBefore = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
integer last = LastActiveElement(mask, esize);

if last < 0 then
    result = operand1;
else
    if !isBefore then
        last = last + 1;
        if last >= elements then last = 0;
    for e = 0 to elements-1
        Elem[result, e, esize] = Elem[operand2, last, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

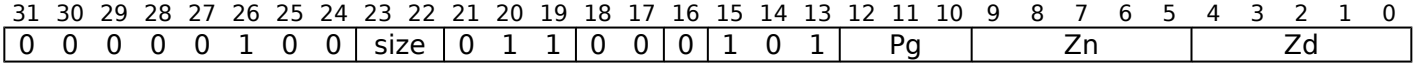
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CLS

Count leading sign bits (predicated)

Count leading sign bits in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.



CLS <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = CountLeadingSignBits(element)<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

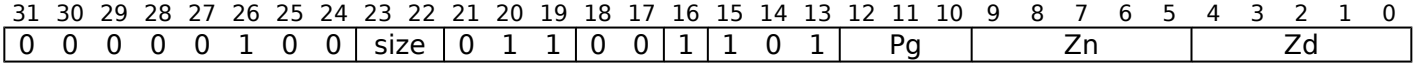
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CLZ

Count leading zero bits (predicated)

Count leading zero bits in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.



CLZ <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = CountLeadingZeroBits(element)<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

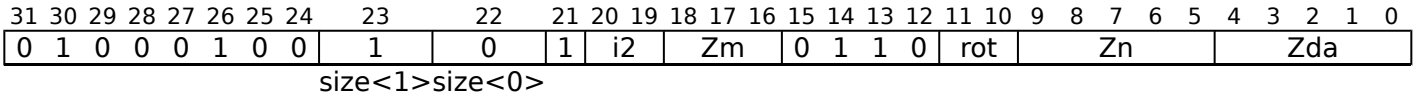
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CMLA (indexed)

Complex integer multiply-add with rotate (indexed)

Multiply the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the integral numbers in each 128-bit segment of the first source vector by the specified complex number in the corresponding the second source vector segment rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation.
Then add the products to the corresponding components of the complex numbers in the addend vector. Destructively place the results in the corresponding elements of the addend vector. This instruction is unpredicated.
These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.
Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.
It has encodings from 2 classes: 16-bit and 32-bit

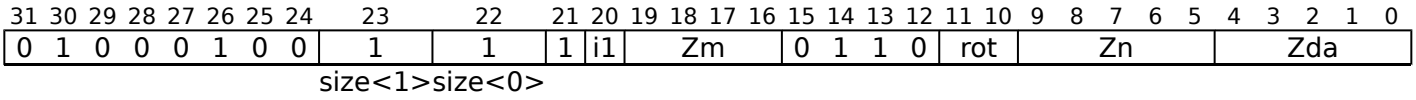
16-bit



CMLA <Zda>.H, <Zn>.H, <Zm>.H[<imm>], <const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_r = (rot<0> != rot<1>);
boolean sub_i = (rot<1> == '1');
```

32-bit



CMLA <Zda>.S, <Zn>.S, <Zm>.S[<imm>], <const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_r = (rot<0> != rot<1>);
boolean sub_i = (rot<1> == '1');
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the 16-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

For the 32-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the 16-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.

For the 32-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

<const> Is the const specifier, encoded in “rot”:

rot	<const>
00	#0
01	#90
10	#180
11	#270

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (2 * esize);
constant integer pairspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for p = 0 to pairs-1
    integer segmentbase = p - (p MOD pairspersegment);
    integer s = segmentbase + index;
    integer elt1_a = SInt(Elem[operand1, 2 * p + sel_a, esize]);
    integer elt2_a = SInt(Elem[operand2, 2 * s + sel_a, esize]);
    integer elt2_b = SInt(Elem[operand2, 2 * s + sel_b, esize]);
    bits(esize) elt3_r = Elem[operand3, 2 * p + 0, esize];
    bits(esize) elt3_i = Elem[operand3, 2 * p + 1, esize];
    integer product_r =  elt1_a * elt2_a;
    integer product_i =  elt1_a * elt2_b;
    if sub_r then
        Elem[result, 2 * p + 0, esize] = elt3_r - product_r;
    else
        Elem[result, 2 * p + 0, esize] = elt3_r + product_r;
    if sub_i then
        Elem[result, 2 * p + 1, esize] = elt3_i - product_i;
    else
        Elem[result, 2 * p + 1, esize] = elt3_i + product_i;

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

CMLA (vectors)

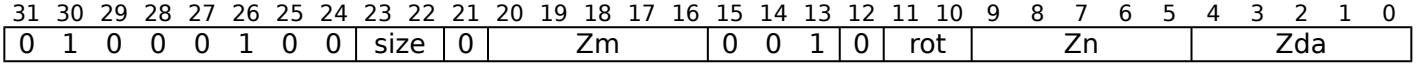
Complex integer multiply-add with rotate

Multiply the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the integral numbers in the first source vector by the corresponding complex number in the second source vector rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation.

Then add the products to the corresponding components of the complex numbers in the addend vector. Destructively place the results in the corresponding elements of the addend vector. This instruction is unpredicated.

These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.



CMLA <Zda>.<T>, <Zn>.<T>, <Zm>.<T>, <const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_r = (rot<0> != rot<1>);
boolean sub_i = (rot<1> == '1');
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<const> Is the const specifier, encoded in “rot”:

rot	<const>
00	#0
01	#90
10	#180
11	#270

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for p = 0 to pairs-1
    integer elt1_a = SInt(Elem[operand1, 2 * p + sel_a, esize]);
    integer elt2_a = SInt(Elem[operand2, 2 * p + sel_a, esize]);
    integer elt2_b = SInt(Elem[operand2, 2 * p + sel_b, esize]);
    bits(esize) elt3_r = Elem[operand3, 2 * p + 0, esize];
    bits(esize) elt3_i = Elem[operand3, 2 * p + 1, esize];
    integer product_r =  elt1_a * elt2_a;
    integer product_i =  elt1_a * elt2_b;
    if sub_r then
        Elem[result, 2 * p + 0, esize] = elt3_r - product_r;
    else
        Elem[result, 2 * p + 0, esize] = elt3_r + product_r;
    if sub_i then
        Elem[result, 2 * p + 1, esize] = elt3_i - product_i;
    else
        Elem[result, 2 * p + 1, esize] = elt3_i + product_i;

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CMP<cc> (immediate)

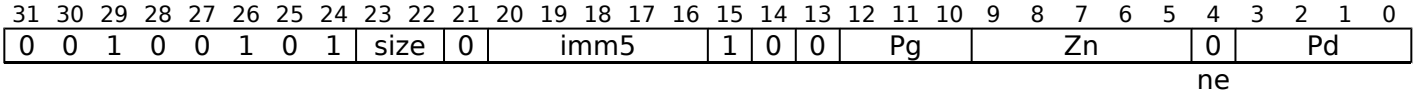
Compare vector to immediate

Compare active integer elements in the source vector with an immediate, and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

<cc>	Comparison
EQ	equal
GE	signed greater than or equal
GT	signed greater than
HI	unsigned higher than
HS	unsigned higher than or same
LE	signed less than or equal
LO	unsigned lower than
LS	unsigned lower than or same
LT	signed less than
NE	not equal

It has encodings from 10 classes: Equal , Greater than , Greater than or equal , Higher , Higher or same , Less than , Less than or equal , Lower , Lower or same and Not equal

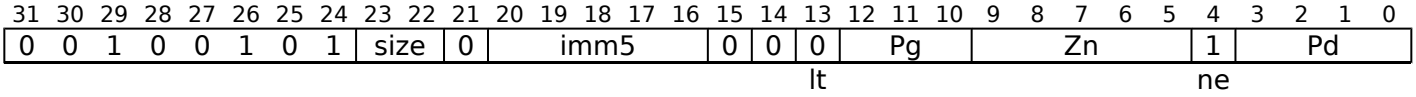
Equal



CMPEQ <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_EQ;
integer imm = SInt(imm5);
boolean unsigned = FALSE;
```

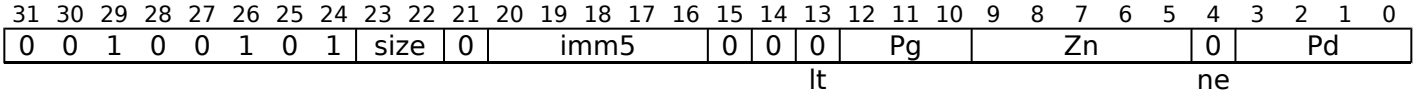
Greater than



CMPGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;
integer imm = SInt(imm5);
boolean unsigned = FALSE;
```

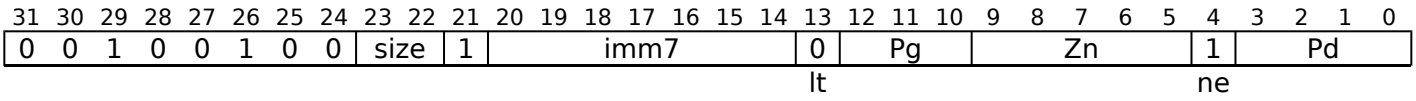
Greater than or equal



CMPGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;
integer imm = SInt(imm5);
boolean unsigned = FALSE;
```

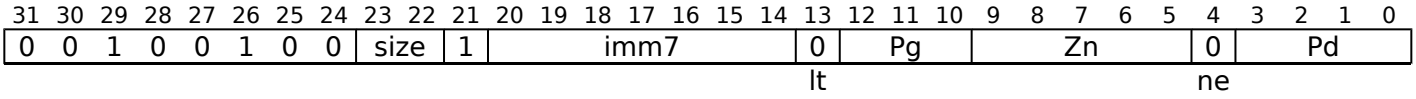
Higher



CMPHI <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;
integer imm = UInt(imm7);
boolean unsigned = TRUE;
```

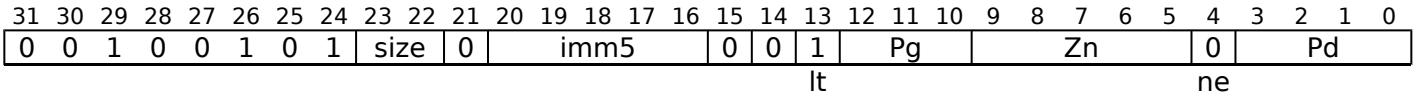
Higher or same



CMPHS <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;
integer imm = UInt(imm7);
boolean unsigned = TRUE;
```

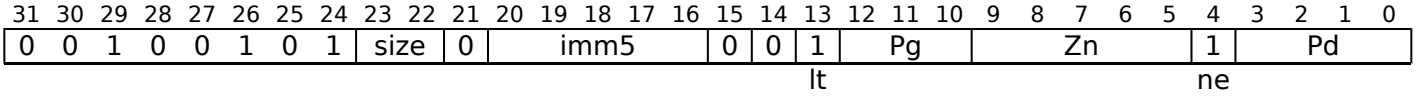
Less than



CMPLT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_LT;
integer imm = SInt(imm5);
boolean unsigned = FALSE;
```

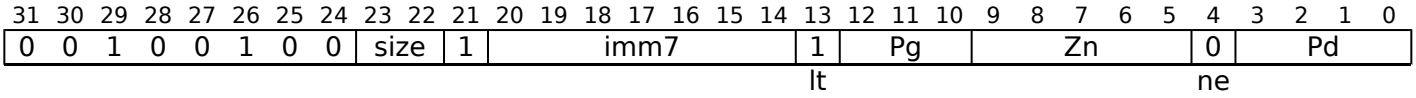
Less than or equal



CMPLE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_LE;
integer imm = SInt(imm5);
boolean unsigned = FALSE;
```

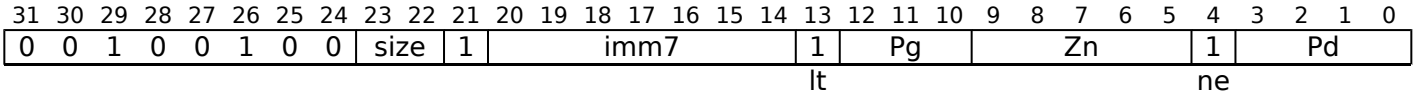
Lower



CMPLO <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_LT;
integer imm = UInt(imm7);
boolean unsigned = TRUE;
```

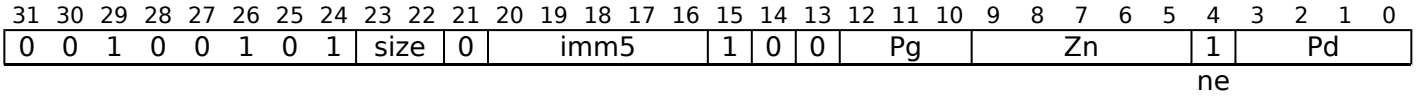
Lower or same



CMPLS <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_LE;
integer imm = UInt(imm7);
boolean unsigned = TRUE;
```

Not equal



CMPNE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #<imm>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_NE;
integer imm = SInt(imm5);
boolean unsigned = FALSE;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<imm> For the equal, greater than, greater than or equal, less than, less than or equal and not equal variant: is the signed immediate operand, in the range -16 to 15, encoded in the "imm5" field.

For the higher, higher or same, lower and lower or same variant: is the unsigned immediate operand, in the range 0 to 127, encoded in the "imm7" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(PL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    if ElemP[mask, e, esize] == '1' then
        boolean cond;
        case op of
            when Cmp_EQ cond = element1 == imm;
            when Cmp_NE cond = element1 != imm;
            when Cmp_GE cond = element1 >= imm;
            when Cmp_LT cond = element1 <  imm;
            when Cmp_GT cond = element1 >  imm;
            when Cmp_LE cond = element1 <= imm;
        ElemP[result, e, esize] = if cond then '1' else '0';
    else
        ElemP[result, e, esize] = '0';

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register or NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CMP<cc> (vectors)

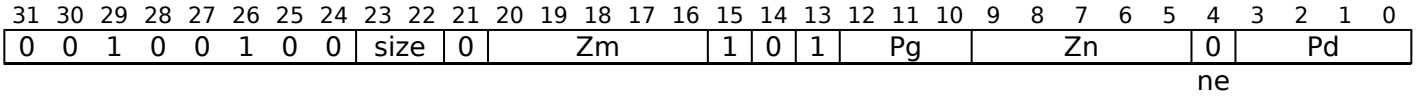
Compare vectors

Compare active integer elements in the first source vector with corresponding elements in the second source vector, and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

<cc>	Comparison
EQ	equal
GE	signed greater than or equal
GT	signed greater than
HI	unsigned higher than
HS	unsigned higher than or same
NE	not equal

This instruction is used by the pseudo-instructions CMPLE (vectors), CMPLO (vectors), CMPLS (vectors), and CMPLT (vectors).
It has encodings from 6 classes: Equal , Greater than , Greater than or equal , Higher , Higher or same and Not equal

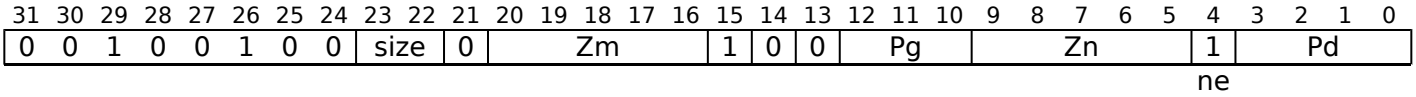
Equal



CMPEQ <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_EQ;
boolean unsigned = FALSE;
```

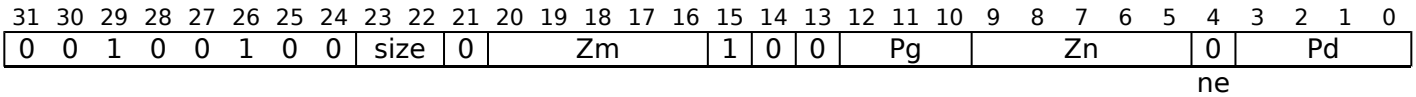
Greater than



CMPGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;
boolean unsigned = FALSE;
```

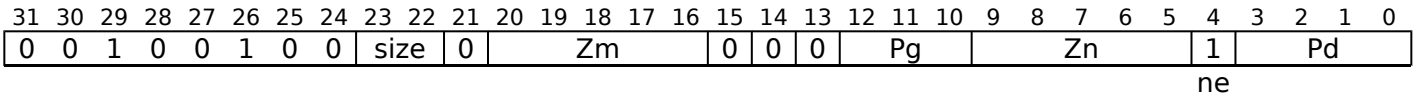
Greater than or equal



CMPGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;
boolean unsigned = FALSE;
```

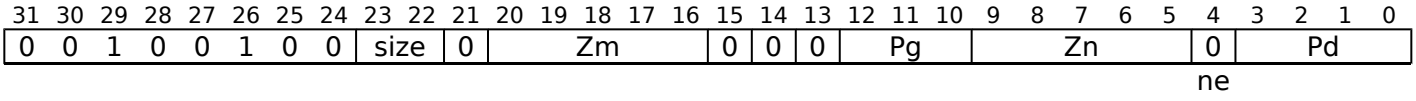
Higher



CMPHI <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;
boolean unsigned = TRUE;
```

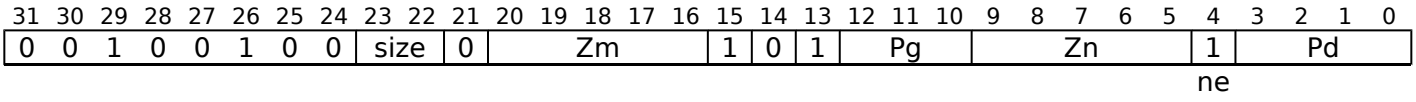
Higher or same



CMPHS <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;
boolean unsigned = TRUE;
```

Not equal



CMPNE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_NE;
boolean unsigned = FALSE;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(PL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    if ElemP[mask, e, esize] == '1' then
        boolean cond;
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        case op of
            when Cmp_EQ cond = element1 == element2;
            when Cmp_NE cond = element1 != element2;
            when Cmp_GE cond = element1 >= element2;
            when Cmp_LT cond = element1 <  element2;
            when Cmp_GT cond = element1 >  element2;
            when Cmp_LE cond = element1 <= element2;
        ElemP[result, e, esize] = if cond then '1' else '0';
    else
        ElemP[result, e, esize] = '0';

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register or NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CMP<cc> (wide elements)

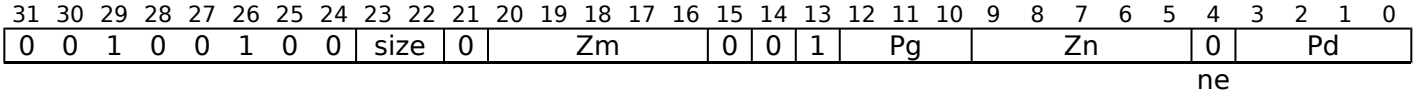
Compare vector to 64-bit wide elements

Compare active integer elements in the first source vector with overlapping 64-bit doubleword elements in the second source vector, and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

<cc>	Comparison
EQ	equal
GE	signed greater than or equal
GT	signed greater than
HI	unsigned higher than
HS	unsigned higher than or same
LE	signed less than or equal
LO	unsigned lower than
LS	unsigned lower than or same
LT	signed less than
NE	not equal

It has encodings from 10 classes: Equal , Greater than , Greater than or equal , Higher , Higher or same , Less than , Less than or equal , Lower , Lower or same and Not equal

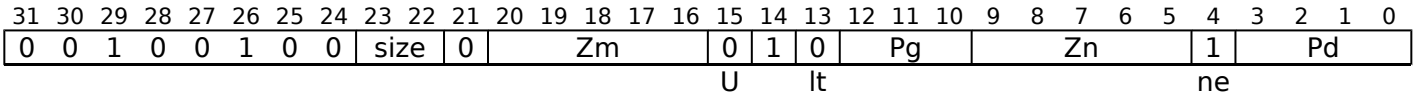
Equal



CMPEQ <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_EQ;
boolean unsigned = FALSE;
```

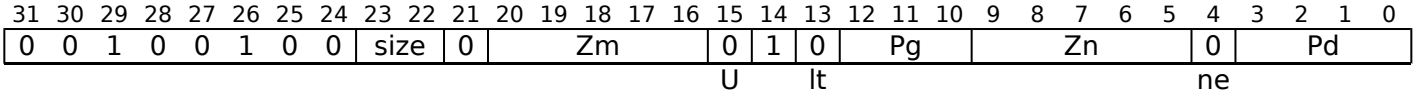
Greater than



CMPGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;
boolean unsigned = FALSE;
```

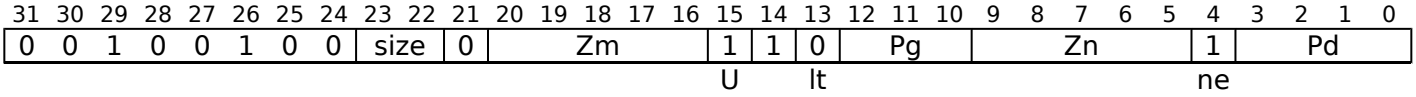
Greater than or equal



CMPGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;
boolean unsigned = FALSE;
```

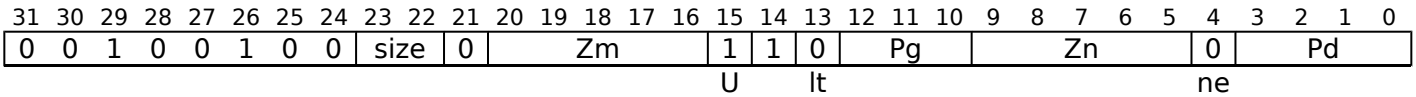
Higher



CMPHI <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;
boolean unsigned = TRUE;
```

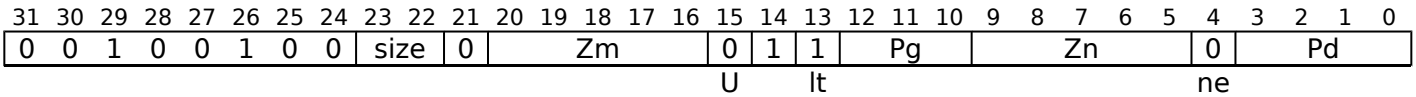
Higher or same



CMPHS <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;
boolean unsigned = TRUE;
```

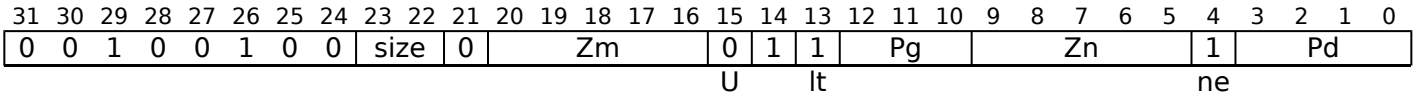
Less than



CMPLT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_LT;
boolean unsigned = FALSE;
```

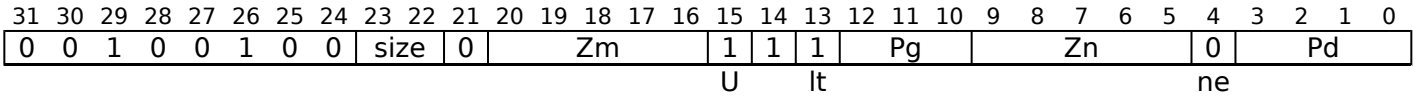
Less than or equal



CMPLE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_LE;
boolean unsigned = FALSE;
```

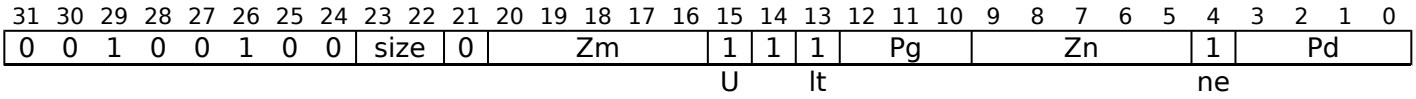
Lower



CMPLO <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_LT;
boolean unsigned = TRUE;
```

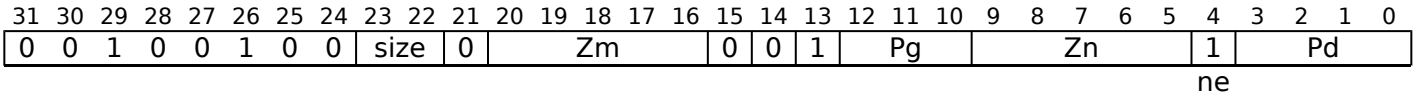
Lower or same



CMPLS <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_LE;
boolean unsigned = TRUE;
```

Not equal



CMPNE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_NE;
boolean unsigned = FALSE;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	RESERVED

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(PL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    if ElemP[mask, e, esize] == '1' then
        boolean cond;
        integer element2 = Int(Elem[operand2, (e * esize) DIV 64, 64], unsigned);
        case op of
            when Cmp_EQ cond = element1 == element2;
            when Cmp_NE cond = element1 != element2;
            when Cmp_GE cond = element1 >= element2;
            when Cmp_LT cond = element1 <  element2;
            when Cmp_GT cond = element1 >  element2;
            when Cmp_LE cond = element1 <= element2;
        ElemP[result, e, esize] = if cond then '1' else '0';
    else
        ElemP[result, e, esize] = '0';

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register or NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

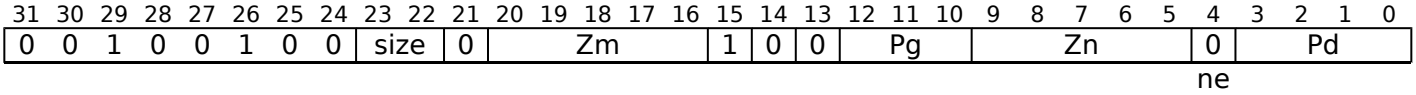
CMPLE (vectors)

Compare signed less than or equal to vector, setting the condition flags

Compare active signed integer elements in the first source vector being less than or equal to corresponding signed elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is a pseudo-instruction of CMP<cc> (vectors). This means:

- The encodings in this description are named to match the encodings of CMP<cc> (vectors).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.



CMPLE <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>

is equivalent to

CMPGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register or NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

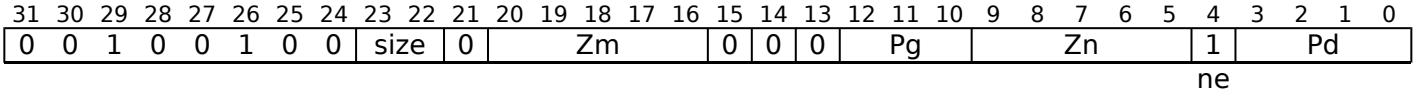
CMPLO (vectors)

Compare unsigned lower than vector, setting the condition flags

Compare active unsigned integer elements in the first source vector being lower than corresponding unsigned elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is a pseudo-instruction of CMP<cc> (vectors). This means:

- The encodings in this description are named to match the encodings of CMP<cc> (vectors).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.



CMPLO <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>

is equivalent to

CMPHI <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register or NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

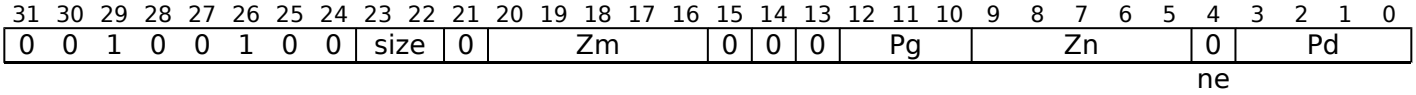
CMPLS (vectors)

Compare unsigned lower or same as vector, setting the condition flags

Compare active unsigned integer elements in the first source vector being lower than or same as corresponding unsigned elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is a pseudo-instruction of CMP<cc> (vectors). This means:

- The encodings in this description are named to match the encodings of CMP<cc> (vectors).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.



CMPLS <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>

is equivalent to

CMPHS <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register or NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

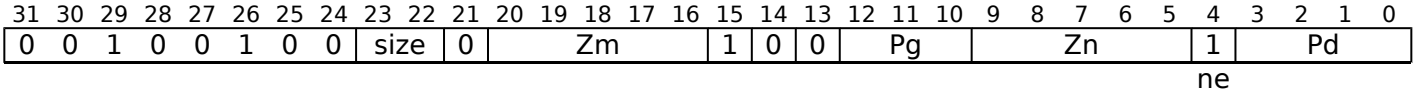
CMPLT (vectors)

Compare signed less than vector, setting the condition flags

Compare active signed integer elements in the first source vector being less than corresponding signed elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is a pseudo-instruction of CMP<cc> (vectors). This means:

- The encodings in this description are named to match the encodings of CMP<cc> (vectors).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.



CMPLT <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>

is equivalent to

CMPGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of CMP<cc> (vectors) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

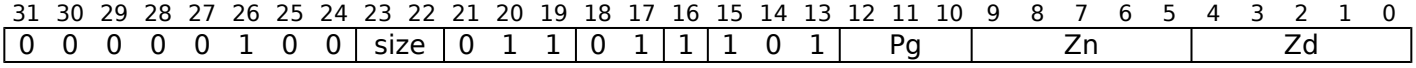
If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register or NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CNOT

Logically invert boolean condition in vector (predicated)

Logically invert the boolean value in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.
Boolean TRUE is any non-zero value in a source, and one in a result element. Boolean FALSE is always zero.



CNOT <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = ZeroExtend(IsZeroBit(element), esize);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

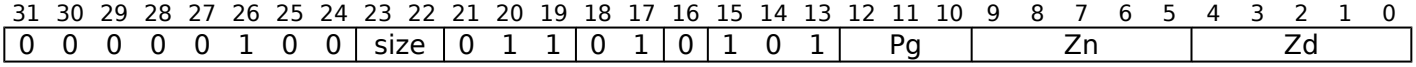
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CNT

Count non-zero bits (predicated)

Count non-zero bits in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.



CNT <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = BitCount(element)<esize-1:0>;

Z[d, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CNTB, CNTD, CNTH, CNTW

Set scalar to multiple of predicate constraint element count

Determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then places the result in the scalar destination.

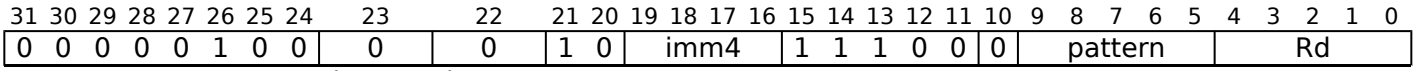
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 4 classes: Byte , Doubleword , Halfword and Word

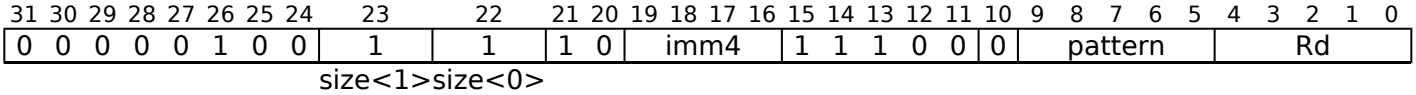
Byte



CNTB <Xd>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer d = UInt(Rd);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

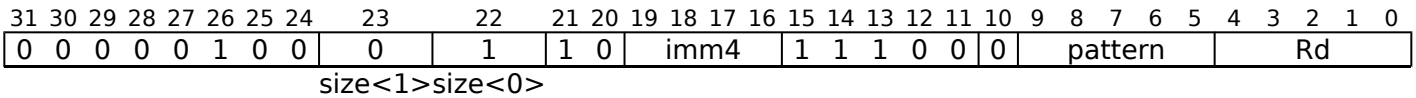
Doubleword



CNTD <Xd>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer d = UInt(Rd);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

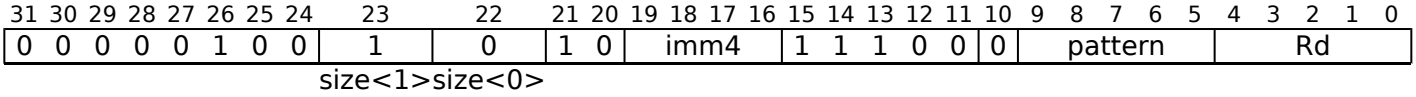
Halfword



CNTH <Xd>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer d = UInt(Rd);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

Word




```
CNTW <Xd>{, <pattern>{, MUL #<imm>}}
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer d = UInt(Rd);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

Assembler Symbols

<Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);

X[d, 64] = (count * imm)<63:0>;
```

Operational information

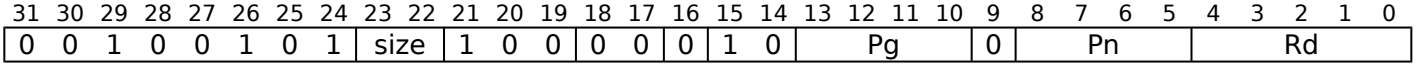
If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

CNTP

Set scalar to count of true predicate elements

Counts the number of active and true elements in the source predicate and places the scalar result in the destination general-purpose register. Inactive predicate elements are not counted.



CNTP <Xd>, <Pg>, <Pn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Pn);
integer d = UInt(Rd);
```

Assembler Symbols

<Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand = P[n, PL];
bits(64) sum = Zeros(64);

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' && ElemP[operand, e, esize] == '1' then
        sum = sum + 1;
X[d, 64] = sum;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

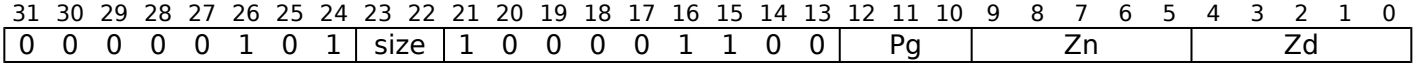
If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

COMPACT

Shuffle active elements of vector to the right and fill with zero

Read the active elements from the source vector and pack them into the lowest-numbered elements of the destination vector. Then set any remaining elements of the destination vector to zero.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.



COMPACT <Zd>.<T>, <Pg>, <Zn>.<T>

```
if !HaveSVE() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size<0>”:

size<0>	<T>
0	S
1	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

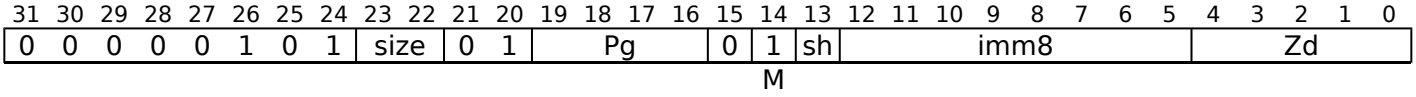
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Zeros(VL);
integer x = 0;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand1, e, esize];
        Elem[result, x, esize] = element;
        x = x + 1;

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CPY (immediate, merging)

Copy signed integer immediate to vector elements (merging)

Copy a signed integer immediate to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.
The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).
The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<simm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".
This instruction is used by the alias MOV (immediate, predicated, merging).
This instruction is used by the pseudo-instruction FMOV (zero, predicated).



CPY <Zd>.<T>, <Pg>/M, #<imm>{, <shift>}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer d = UInt(Zd);
boolean merging = TRUE;
integer imm = SInt(imm8);
if sh == '1' then imm = imm << 8;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<imm> Is a signed immediate in the range -128 to 127, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #8

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) dest = Z[d, VL];
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = imm<esize-1:0>;
    elsif merging then
        Elem[result, e, esize] = Elem[dest, e, esize];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CPY (immediate, zeroing)

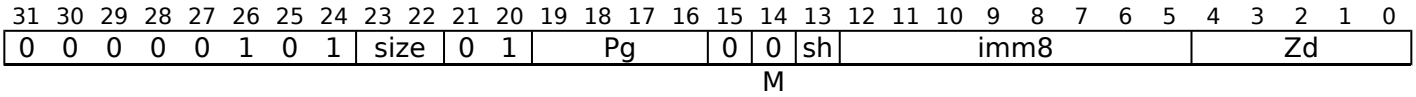
Copy signed integer immediate to vector elements (zeroing)

Copy a signed integer immediate to each active element in the destination vector. Inactive elements in the destination vector register are set to zero.

The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<simm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

This instruction is used by the alias MOV (immediate, predicated, zeroing).



CPY <Zd>.<T>, <Pg>/Z, #<imm>{, <shift>}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer d = UInt(Zd);
boolean merging = FALSE;
integer imm = SInt(imm8);
if sh == '1' then imm = imm << 8;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<imm> Is a signed immediate in the range -128 to 127, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #8

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) dest = Z[d, VL];
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = imm<esize-1:0>;
    elsif merging then
        Elem[result, e, esize] = Elem[dest, e, esize];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

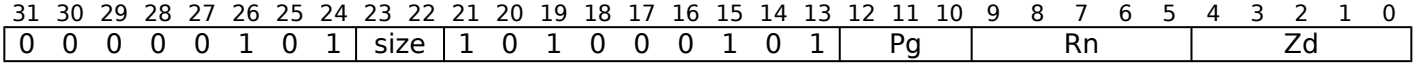
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CPY (scalar)

Copy general-purpose register to vector elements (predicated)

Copy the general-purpose scalar source register to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.
This instruction is used by the alias MOV (scalar, predicated).



CPY <Zd>.<T>, <Pg>/M, <R><n|SP>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Rn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<R> Is a width specifier, encoded in “size”:

size	<R>
01	W
x0	W
11	X

<n|SP> Is the number [0-30] of the general-purpose source register or the name SP (31), encoded in the "Rn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) result = Z[d, VL];
if AnyActiveElement(mask, esize) then
    bits(64) operand1;
    if n == 31 then
        operand1 = SP[];
    else
        operand1 = X[n, 64];

    for e = 0 to elements-1
        if ElemP[mask, e, esize] == '1' then
            Elem[result, e, esize] = operand1<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

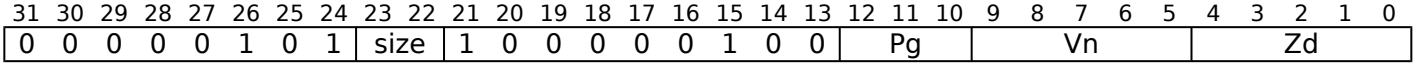
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CPY (SIMD&FP scalar)

Copy SIMD&FP scalar register to vector elements (predicated)

Copy the SIMD & floating-point scalar source register to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.
This instruction is used by the alias MOV (SIMD&FP scalar, predicated).



CPY <Zd>.<T>, <Pg>/M, <V><n>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Vn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<n> Is the number [0-31] of the source SIMD&FP register, encoded in the "Vn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(esize) operand1 = if AnyActiveElement(mask, esize) then V[n, esize] else Zeros(esize);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = operand1;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

CTERMEQ, CTERMNE

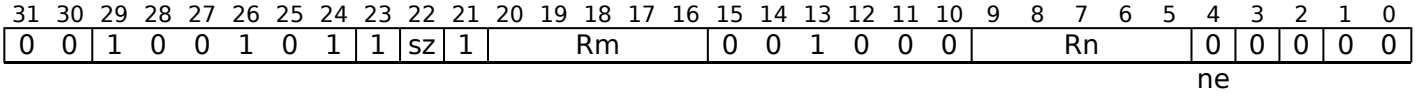
Compare and terminate loop

Detect termination conditions in serialized vector loops. Tests whether the comparison between the scalar source operands holds true and if not tests the state of the !LAST condition flag (C) which indicates whether the previous flag-setting predicate instruction selected the last element of the vector partition.
The Z and C condition flags are preserved by this instruction. The N and V condition flags are set as a pair to generate one of the following conditions for a subsequent conditional instruction:

Condition	N	V	Meaning
GE	0	0	Continue loop (compare failed and last element not selected)
LT	0	1	Terminate loop (last element selected)
LT	1	0	Terminate loop (compare succeeded)
GE	1	1	Never generated

The scalar source operands are 32-bit or 64-bit general-purpose registers of the same size.
It has encodings from 2 classes: Equal and Not equal

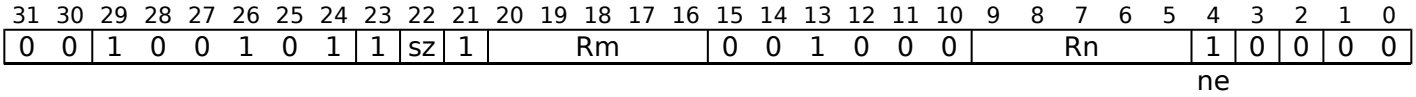
Equal



CTERMEQ <R><n>, <R><m>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer n = UInt(Rn);
integer m = UInt(Rm);
SVECmp op = Cmp_EQ;
```

Not equal



CTERMNE <R><n>, <R><m>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer n = UInt(Rn);
integer m = UInt(Rm);
SVECmp op = Cmp_NE;
```

Assembler Symbols

<R> Is a width specifier, encoded in “sz”:

sz	<R>
0	W
1	X

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
bits(esize) operand1 = X[n, esize];
bits(esize) operand2 = X[m, esize];
integer element1 = UInt(operand1);
integer element2 = UInt(operand2);
boolean term;

case op of
    when Cmp_EQ term = element1 == element2;
    when Cmp_NE term = element1 != element2;
if term then
    PSTATE.N = '1';
    PSTATE.V = '0';
else
    PSTATE.N = '0';
    PSTATE.V = (NOT PSTATE.C);
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

DECB, DECD, DECH, DECW (scalar)

Decrement scalar by multiple of predicate constraint element count

Determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination.

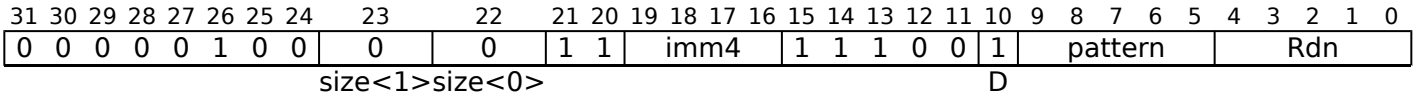
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 4 classes: Byte , Doubleword , Halfword and Word

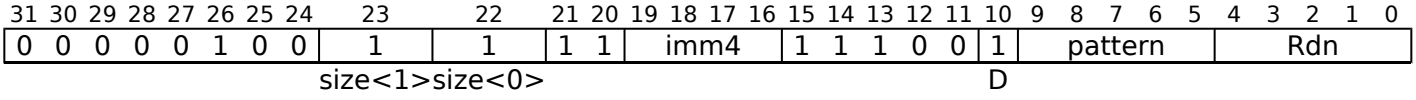
Byte



DECB <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

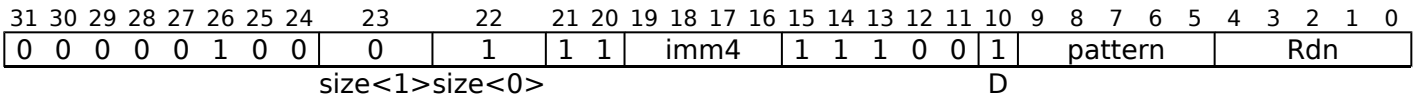
Doubleword



DECD <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

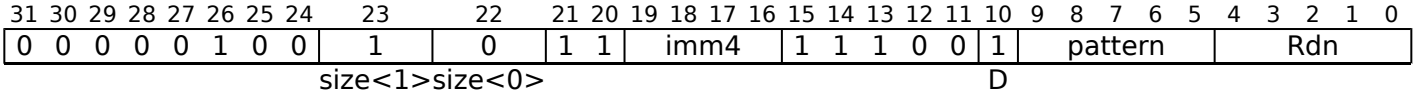
Halfword



DECH <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

Word




```
DECW <Xdn>{, <pattern>{, MUL #<imm>}}
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

Assembler Symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
constant integer VL = CurrentVL;
bits(64) operand1 = X[dn, 64];

X[dn, 64] = operand1 - (count * imm);
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

DECD, DECH, DECW (vector)

Decrement vector by multiple of predicate constraint element count

Determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements.

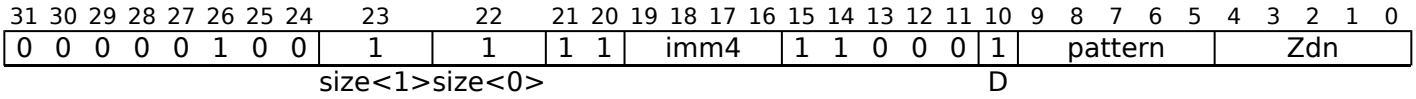
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 3 classes: Doubleword , Halfword and Word

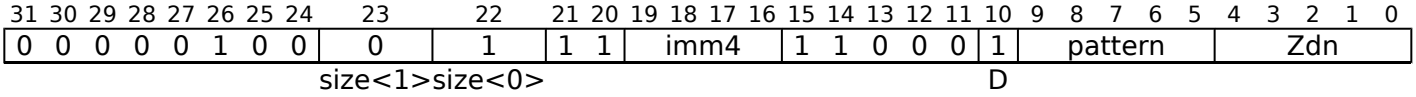
Doubleword



DECD <Zdn>.D{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

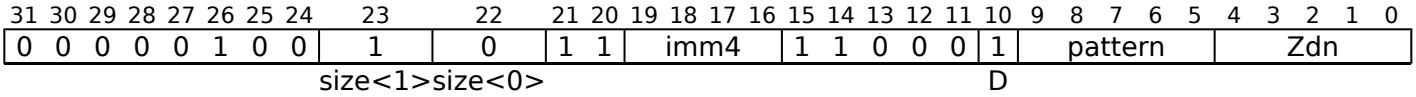
Halfword



DECH <Zdn>.H{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

Word



DECW <Zdn>.S{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    Elem[result, e, esize] = Elem[operand1, e, esize] - (count * imm);

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

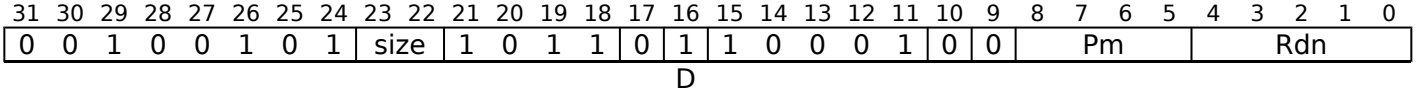
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

DECP (scalar)

Decrement scalar by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to decrement the scalar destination.



DECP <Xdn>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
```

Assembler Symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) operand1 = X[dn, 64];
bits(PL) operand2 = P[m, PL];
integer count = 0;

for e = 0 to elements-1
    if ElemP[operand2, e, esize] == '1' then
        count = count + 1;

X[dn, 64] = operand1 - count;
```

Operational information

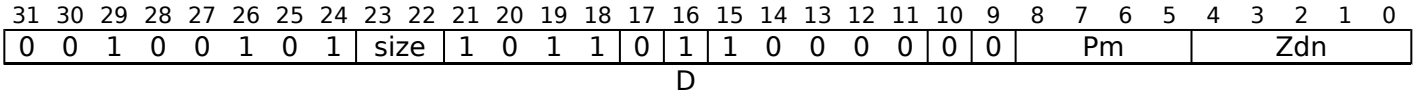
If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

DECP (vector)

Decrement vector by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to decrement all destination vector elements.
The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.



DECP <Zdn>.<T>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) operand2 = P[m, PL];
bits(VL) result;
integer count = 0;

for e = 0 to elements-1
    if ElemP[operand2, e, esize] == '1' then
        count = count + 1;

for e = 0 to elements-1
    Elem[result, e, esize] = Elem[operand1, e, esize] - count;

Z[dn, VL] = result;
```

Operational information

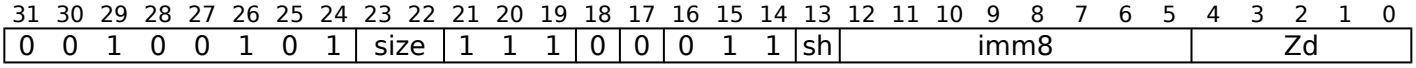
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

DUP (immediate)

Broadcast signed immediate to vector elements (unpredicated)

Unconditionally broadcast the signed integer immediate into each element of the destination vector. This instruction is unpredicated.
The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).
The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<simm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".
This instruction is used by the alias MOV (immediate, unpredicated).
This instruction is used by the pseudo-instruction FMOV (zero, unpredicated).



DUP <Zd>.<T>, #<imm>{, <shift>}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer d = UInt(Zd);
integer imm = SInt(imm8);
if sh == '1' then imm = imm << 8;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<imm> Is a signed immediate in the range -128 to 127, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #8

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(VL) result = Replicate(imm<esize-1:0>, VL DIV esize);
Z[d, VL] = result;
```

Operational information

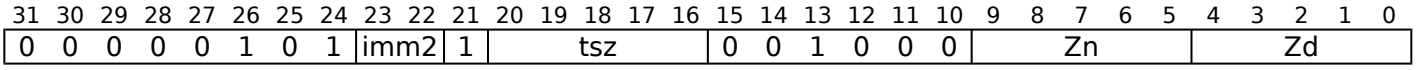
If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

DUP (indexed)

Broadcast indexed element to vector (unpredicated)

Unconditionally broadcast the indexed source vector element into each element of the destination vector. This instruction is unpredicated.
The immediate element index is in the range of 0 to 63 (bytes), 31 (halfwords), 15 (words), 7 (doublewords) or 3 (quadwords). Selecting an element beyond the accessible vector length causes the destination vector to be set to zero.
This instruction is used by the alias MOV (SIMD&FP scalar, unpredicated).



DUP <Zd>.<T>, <Zn>.<T>[<imm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
bits(7) imm = imm2:tsz;
integer esize;
integer index;
case tsz of
    when '00000' UNDEFINED;
    when '10000' esize = 128; index = UInt(imm<6:5>);
    when 'x1000' esize = 64;  index = UInt(imm<6:4>);
    when 'xx100' esize = 32;  index = UInt(imm<6:3>);
    when 'xxx10' esize = 16;  index = UInt(imm<6:2>);
    when 'xxxx1' esize = 8;   index = UInt(imm<6:1>);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tsz”:

tsz	<T>
00000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D
10000	Q

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<imm> Is the immediate index, in the range 0 to one less than the number of elements in 512 bits, encoded in "imm2:tsz".

Alias Conditions

Alias	Is preferred when
MOV (SIMD&FP scalar, unpredicated)	`BitCount(imm2:tsz) == 1`
MOV (SIMD&FP scalar, unpredicated)	`BitCount(imm2:tsz) > 1`

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) result;
bits(esize) element;

if index >= elements then
    element = Zeros(esize);
else
    element = Elem[operand1, index, esize];
result = Replicate(element, VL DIV esize);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

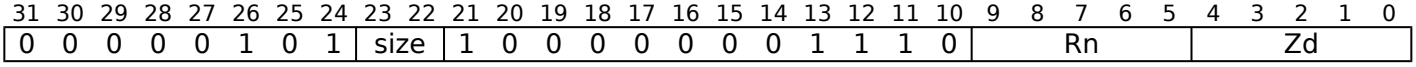
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

DUP (scalar)

Broadcast general-purpose register to vector elements (unpredicated)

Unconditionally broadcast the general-purpose scalar source register into each element of the destination vector. This instruction is unpredicated.

This instruction is used by the alias MOV (scalar, unpredicated).



DUP <Zd>.<T>, <R><n|SP>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Rn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<R> Is a width specifier, encoded in “size”:

size	<R>
01	W
x0	W
11	X

<n|SP> Is the number [0-30] of the general-purpose source register or the name SP (31), encoded in the "Rn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer PL = VL DIV 8;
bits(64) operand;
if n == 31 then
    operand = SP[];
else
    operand = X[n, 64];
bits(VL) result;

for e = 0 to elements-1
    Elem[result, e, esize] = operand<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.

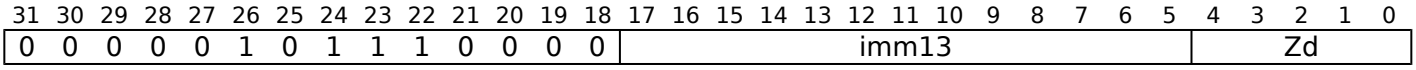
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

DUPM

Broadcast logical bitmask immediate to vector (unpredicated)

Unconditionally broadcast the logical bitmask immediate into each element of the destination vector. This instruction is unpredicated. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits.
This instruction is used by the alias MOV.



DUPM <Zd>.<T>, #<const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer d = UInt(Zd);
bits(esize) imm;
(imm, -) = DecodeBitMasks(imm13<12>, imm13<5:0>, imm13<11:6>, TRUE, esize);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “imm13<12>:imm13<5:0>”:

imm13<12>	imm13<5:0>	<T>
0	0xxxxx	S
0	10xxxx	H
0	110xxx	B
0	1110xx	B
0	11110x	B
0	111110	RESERVED
0	111111	RESERVED
1	xxxxxx	D

<const> Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the "imm13" field.

Alias Conditions

Alias	Is preferred when
MOV	SVEMoveMaskPreferred(imm13)

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) result = Replicate(imm, VL DIV esize);
Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

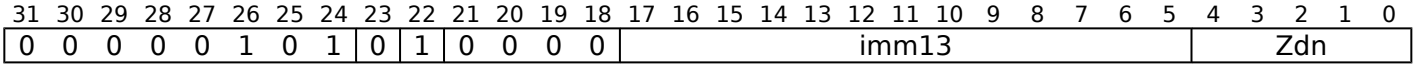
EON

Bitwise exclusive OR with inverted immediate (unpredicated)

Bitwise exclusive OR an inverted immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This is a pseudo-instruction of EOR (immediate). This means:

- The encodings in this description are named to match the encodings of EOR (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of EOR (immediate) gives the operational pseudocode for this instruction.



EON <Zdn>.<T>, <Zdn>.<T>, #<const>

is equivalent to

EOR <Zdn>.<T>, <Zdn>.<T>, #(-<const> - 1)

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “imm13<12>:imm13<5:0>”:

imm13<12>	imm13<5:0>	<T>
0	0xxxxx	S
0	10xxxx	H
0	110xxx	B
0	1110xx	B
0	11110x	B
0	111110	RESERVED
0	111111	RESERVED
1	xxxxxx	D

<const> Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the "imm13" field.

Operation

The description of EOR (immediate) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

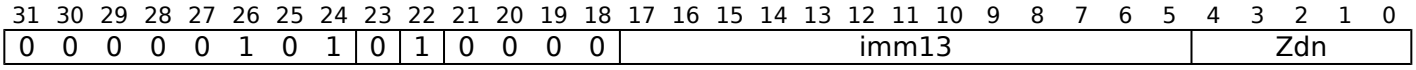
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

EOR (immediate)

Bitwise exclusive OR with immediate (unpredicated)

Bitwise exclusive OR an immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This instruction is used by the pseudo-instruction EON.



EOR <Zdn>.<T>, <Zdn>.<T>, #<const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer dn = UInt(Zdn);
bits(64) imm;
(imm, -) = DecodeBitMasks(imm13<12>, imm13<5:0>, imm13<11:6>, TRUE, 64);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “imm13<12>:imm13<5:0>”:

imm13<12>	imm13<5:0>	<T>
0	0xxxxx	S
0	10xxxx	H
0	110xxx	B
0	1110xx	B
0	11110x	B
0	111110	RESERVED
0	111111	RESERVED
1	xxxxxx	D

<const> Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the "imm13" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 64;
bits(VL) operand = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(64) element1 = Elem[operand, e, 64];
    Elem[result, e, 64] = element1 EOR imm;

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

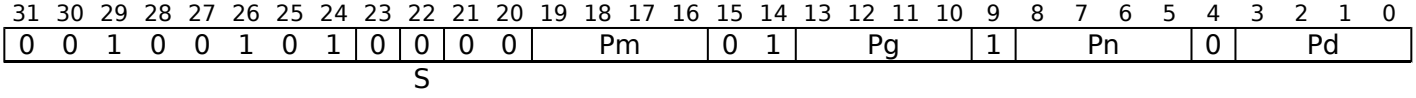
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

EOR (predicates)

Bitwise exclusive OR predicates

Bitwise exclusive OR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.
This instruction is used by the alias NOT (predicate).



EOR <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm>	Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Alias Conditions

Alias	Is preferred when
NOT (predicate)	`Pm == Pg`

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 EOR element2;
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

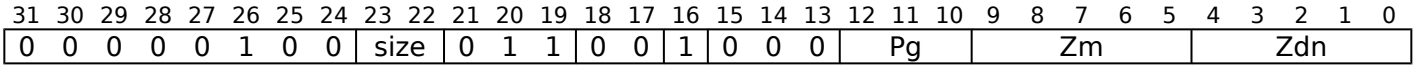
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

EOR (vectors, predicated)

Bitwise exclusive OR vectors (predicated)

Bitwise exclusive OR active elements of the second source vector with corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



EOR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = element1 EOR element2;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

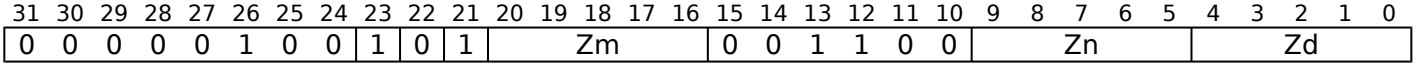
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

EOR (vectors, unpredicated)

Bitwise exclusive OR vectors (unpredicated)

Bitwise exclusive OR all elements of the second source vector with corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.



EOR <Zd>.D, <Zn>.D, <Zm>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];

Z[d, VL] = operand1 EOR operand2;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

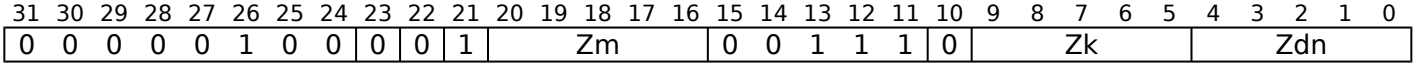
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

EOR3

Bitwise exclusive OR of three vectors

Bitwise exclusive OR the corresponding elements of all three source vectors, and destructively place the results in the corresponding elements of the destination and first source vector. This instruction is unpredicated.



EOR3 <Zdn>.D, <Zdn>.D, <Zm>.D, <Zk>.D

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
integer m = UInt(Zm);
integer k = UInt(Zk);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn>	Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.
<Zk>	Is the name of the third source scalable vector register, encoded in the "Zk" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[k, VL];

Z[dn, VL] = operand1 EOR operand2 EOR operand3;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

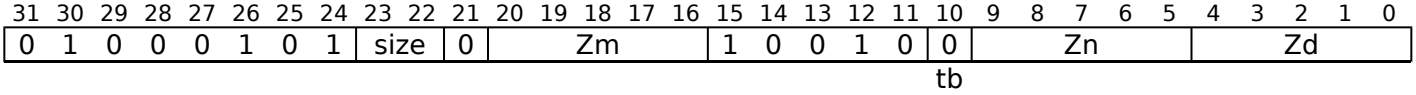
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

EORBT

Interleaving exclusive OR (bottom, top)

Interleaving exclusive OR between the even-numbered elements of the first source vector register and the odd-numbered elements of the second source vector register, placing the result in the even-numbered elements of the destination vector, leaving the odd-numbered elements unchanged. This instruction is unpredicated.



EORBT <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 0;
integer sel2 = 1;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, 2*e + sel1, esize];
    bits(esize) element2 = Elem[operand2, 2*e + sel2, esize];
    Elem[result, 2*e + sel1, esize] = element1 EOR element2;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

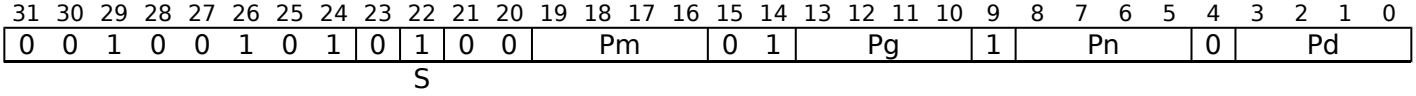
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

EORS

Bitwise exclusive OR predicates, setting the condition flags

Bitwise exclusive OR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.
This instruction is used by the alias NOTS.



EORS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm>	Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Alias Conditions

Alias	Is preferred when
NOTS	`Pm == Pg`

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 EOR element2;
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

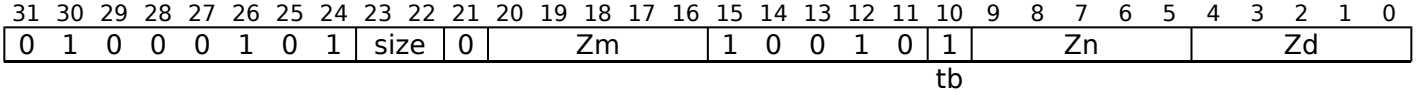
If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

EORTB

Interleaving exclusive OR (top, bottom)

Interleaving exclusive OR between the odd-numbered elements of the first source vector register and the even-numbered elements of the second source vector register, placing the result in the odd-numbered elements of the destination vector, leaving the even-numbered elements unchanged. This instruction is unpredicated.



EORTB <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 1;
integer sel2 = 0;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, 2*e + sel1, esize];
    bits(esize) element2 = Elem[operand2, 2*e + sel2, esize];
    Elem[result, 2*e + sel1, esize] = element1 EOR element2;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

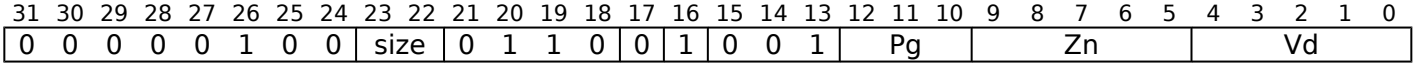
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

EORV

Bitwise exclusive OR reduction to scalar

Bitwise exclusive OR horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as zero.



EORV <V><d>, <Pg>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(esize) result = Zeros(esize);

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        result = result EOR Elem[operand, e, esize];

V[d, esize] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

EXT

Extract vector from pair of vectors

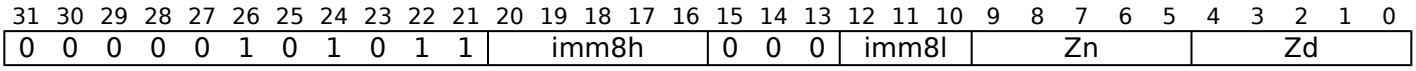
Copy the indexed byte up to the last byte of the first source vector to the bottom of the result vector, then fill the remainder of the result starting from the first byte of the second source vector. The result is placed destructively in the destination and first source vector, or constructively in the destination vector. This instruction is unpredicated.

An index that is greater than or equal to the vector length in bytes is treated as zero, resulting in the first source vector being copied to the result unchanged.

The Destructive encoding of this instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE: The MOVPRFX instruction must be unpredicated. The MOVPRFX instruction must specify the same destination register as this instruction. The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

It has encodings from 2 classes: Constructive and Destructive

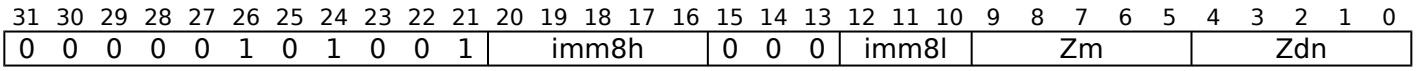
Constructive



EXT <Zd>.B, { <Zn1>.B, <Zn2>.B }, #<imm>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dst = UInt(Zd);
integer s1 = UInt(Zn);
integer s2 = (s1 + 1) MOD 32;
integer position = UInt(imm8h:imm8l);
```

Destructive



EXT <Zdn>.B, <Zdn>.B, <Zm>.B, #<imm>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dst = UInt(Zdn);
integer s1 = dst;
integer s2 = UInt(Zm);
integer position = UInt(imm8h:imm8l);
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn1>	Is the name of the first scalable vector register of a multi-vector sequence, encoded in the "Zn" field.
<Zn2>	Is the name of the second scalable vector register of a multi-vector sequence, encoded in the "Zn" field.
<Zdn>	Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.
<imm>	Is the unsigned immediate operand, in the range 0 to 255, encoded in the "imm8h:imm8l" fields.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[s1, VL];
bits(VL) operand2 = Z[s2, VL];
bits(VL) result;

if position >= elements then
    position = 0;

position = position << 3;
bits(VL*2) concat = operand2 : operand1;
result = concat<(position+VL)-1:position>;

Z[dst, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

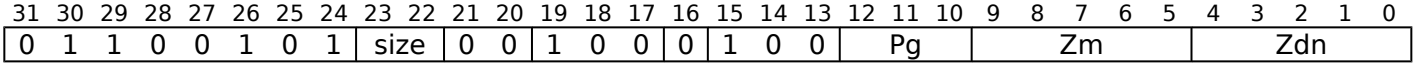
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FABD

Floating-point absolute difference (predicated)

Compute the absolute difference of active floating-point elements of the second source vector and corresponding floating-point elements of the first source vector and destructively place the result in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



FABD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = FPAbs(FPSub(element1, element2, FPCR[]));
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

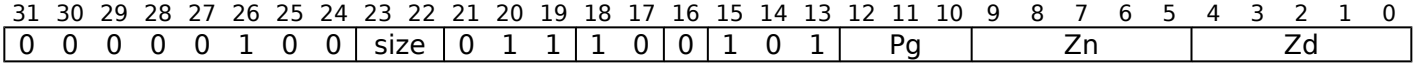
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FABS

Floating-point absolute value (predicated)

Take the absolute value of each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. This clears the sign bit and cannot signal a floating-point exception. Inactive elements in the destination vector register remain unmodified.



FABS <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = FPAbs(element);

Z[d, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

FAC<cc>

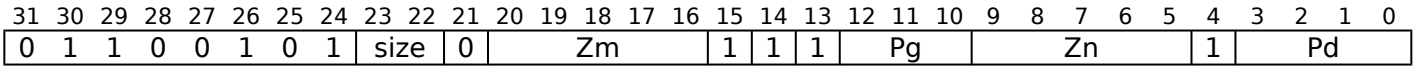
Floating-point absolute compare vectors

Compare active absolute values of floating-point elements in the first source vector with corresponding absolute values of elements in the second source vector, and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

<cc>	Comparison
GE	greater than or equal
GT	greater than
LE	less than or equal
LT	less than

This instruction is used by the pseudo-instructions FACLE, and FACLT.
It has encodings from 2 classes: Greater than and Greater than or equal

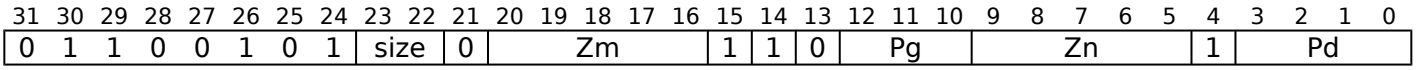
Greater than



FACGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;
```

Greater than or equal



FACGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(PL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        boolean res;
        case op of
            when Cmp_GE res = FPCompareGE(FPAbs(element1), FPAbs(element2), FPCR[]);
            when Cmp_GT res = FPCompareGT(FPAbs(element1), FPAbs(element2), FPCR[]);
        ElemP[result, e, esize] = if res then '1' else '0';
    else
        ElemP[result, e, esize] = '0';

P[d, PL] = result;
```

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

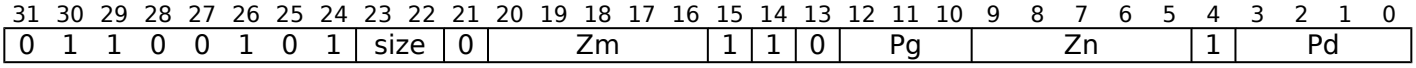
FACLE

Floating-point absolute compare less than or equal

Compare active absolute values of floating-point elements in the first source vector being less than or equal to corresponding absolute values of elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is a pseudo-instruction of FAC<cc>. This means:

- The encodings in this description are named to match the encodings of FAC<cc>.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of FAC<cc> gives the operational pseudocode for this instruction.



FACLE <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>

is equivalent to

FACGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of FAC<cc> gives the operational pseudocode for this instruction.

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

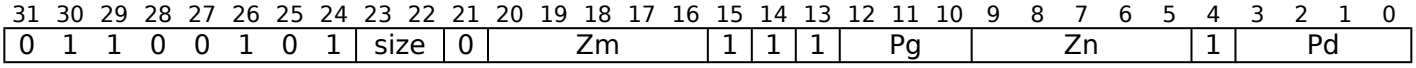
FACLT

Floating-point absolute compare less than

Compare active absolute values of floating-point elements in the first source vector being less than corresponding absolute values of elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is a pseudo-instruction of FAC<cc>. This means:

- The encodings in this description are named to match the encodings of FAC<cc>.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of FAC<cc> gives the operational pseudocode for this instruction.



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size		0	Zm					1	1	1	Pg			Zn					1	Pd			

FACLT <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>

is equivalent to

FACGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of FAC<cc> gives the operational pseudocode for this instruction.

Operational information

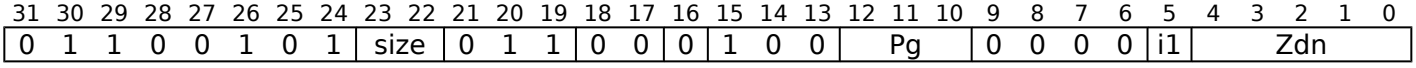
If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FADD (immediate)

Floating-point add immediate (predicated)

Add an immediate to each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.5 or +1.0 only. Inactive elements in the destination vector register remain unmodified.



FADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then FPPointFive('0', esize) else FPOne('0', esize);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the floating-point immediate value, encoded in “i1”:

i1	<const>
0	#0.5
1	#1.0

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = FPAdd(element1, imm, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

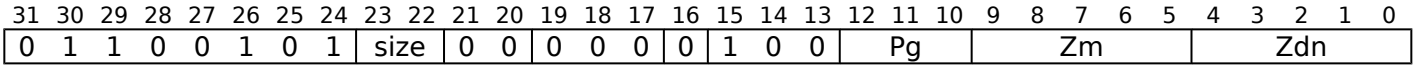
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FADD (vectors, predicated)

Floating-point add vector (predicated)

Add active floating-point elements of the second source vector to corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



FADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = FPAdd(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

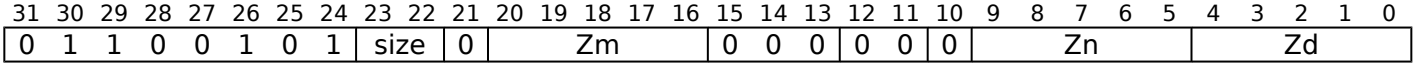
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FADD (vectors, unpredicated)

Floating-point add vector (unpredicated)

Add all floating-point elements of the second source vector to corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.



FADD <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

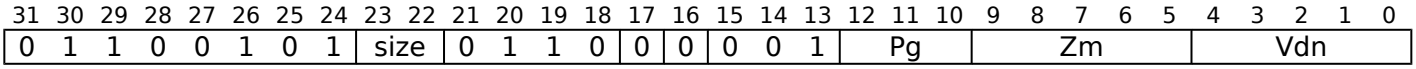
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPAdd(element1, element2, FPCR[]);

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FADDA

Floating-point add strictly-ordered reduction, accumulating in scalar

Floating-point add a SIMD&FP scalar source and all active lanes of the vector source and place the result destructively in the SIMD&FP scalar source register. Vector elements are processed strictly in order from low to high, with the scalar source providing the initial value. Inactive elements in the source vector are ignored.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.



FADDA <V><dn>, <Pg>, <V><dn>, <Zm>.<T>

```
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Vdn);
integer m = UInt(Zm);
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	D

<dn> Is the number [0-31] of the source and destination SIMD&FP register, encoded in the "Vdn" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the source scalable vector register, encoded in the "Zm" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

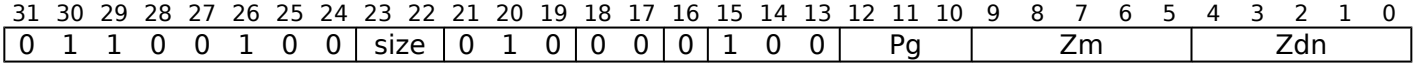
Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(esize) operand1 = V[dn, esize];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(esize) result = operand1;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand2, e, esize];
        result = FPAdd(result, element, FPCR[]);

V[dn, esize] = result;
```

FADDP

Floating-point add pairwise

Add pairs of adjacent floating-point elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.



FADDP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result = Z[dn, VL];
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        if IsEven(e) then
            element1 = Elem[operand1, e + 0, esize];
            element2 = Elem[operand1, e + 1, esize];
        else
            element1 = Elem[operand2, e - 1, esize];
            element2 = Elem[operand2, e + 0, esize];
        Elem[result, e, esize] = FPAdd(element1, element2, FPCR[]);

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

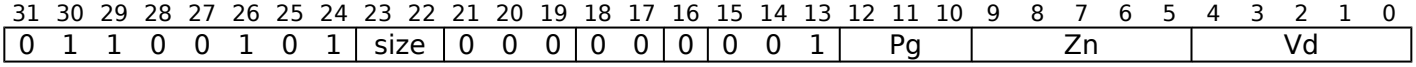
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FADDV

Floating-point add recursive reduction to scalar

Floating-point add horizontally over all lanes of a vector using a recursive pairwise reduction, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as +0.0.



FADDV <V><d>, <Pg>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	D

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(esize) identity = FPZero('0', esize);

V[d, esize] = ReducePredicated(ReduceOp_FADD, operand, mask, identity);
```

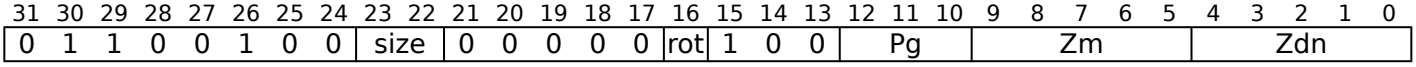
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCADD

Floating-point complex add with rotate (predicated)

Add the real and imaginary components of the active floating-point complex numbers from the first source vector to the complex numbers from the second source vector which have first been rotated by 90 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation, equivalent to multiplying the complex numbers in the second source vector by ±J beforehand. Destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	size		0	0	0	0	0	rot	1	0	0	Pg			Zm					Zdn				

FCADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>, <const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean sub_i = (rot == '0');
boolean sub_r = (rot == '1');
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<const> Is the const specifier, encoded in “rot”:

rot	<const>
0	#90
1	#270

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (2 * esize);
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for p = 0 to pairs-1
    acc_r = Elem[operand1, 2 * p + 0, esize];
    acc_i = Elem[operand1, 2 * p + 1, esize];
    if ElemP[mask, 2 * p + 0, esize] == '1' then
        elt2_i = Elem[operand2, 2 * p + 1, esize];
        if sub_i then elt2_i = FPNeg(elt2_i);
        acc_r = FPAdd(acc_r, elt2_i, FPCR[]);
    if ElemP[mask, 2 * p + 1, esize] == '1' then
        elt2_r = Elem[operand2, 2 * p + 0, esize];
        if sub_r then elt2_r = FPNeg(elt2_r);
        acc_i = FPAdd(acc_i, elt2_r, FPCR[]);
    Elem[result, 2 * p + 0, esize] = acc_r;
    Elem[result, 2 * p + 1, esize] = acc_i;

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCM<cc> (vectors)

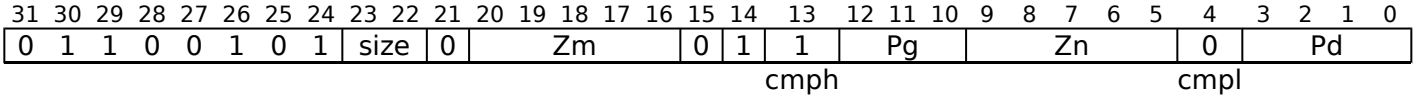
Floating-point compare vectors

Compare active floating-point elements in the first source vector with corresponding elements in the second source vector, and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

<cc>	Comparison
EQ	equal
GE	greater than or equal
GT	greater than
NE	not equal
UO	unordered

This instruction is used by the pseudo-instructions FCMLE (vectors), and FCMLT (vectors).
It has encodings from 5 classes: Equal , Greater than , Greater than or equal , Not equal and Unordered

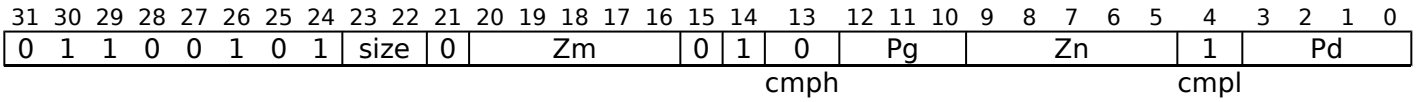
Equal



FCMEQ <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_EQ;
```

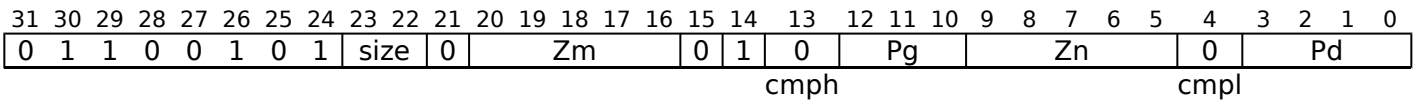
Greater than



FCMGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;
```

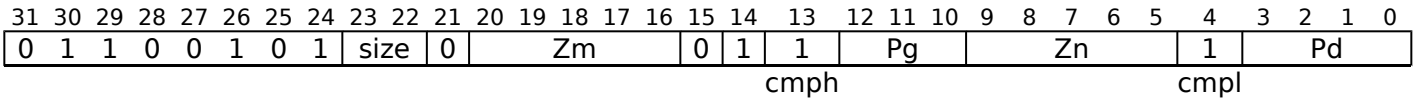
Greater than or equal



FCMGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;
```

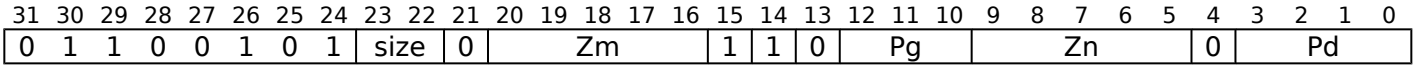
Not equal



FCMNE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_NE;
```

Unordered



FCMUO <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Pd);
SVECmp op = Cmp_UN;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(PL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        boolean res;
        case op of
            when Cmp_EQ res = FPCompareEQ(element1, element2, FPCR[]);
            when Cmp_GE res = FPCompareGE(element1, element2, FPCR[]);
            when Cmp_GT res = FPCompareGT(element1, element2, FPCR[]);
            when Cmp_UN res = FPCompareUN(element1, element2, FPCR[]);
            when Cmp_NE res = FPCompareNE(element1, element2, FPCR[]);
            when Cmp_LT res = FPCompareGT(element2, element1, FPCR[]);
            when Cmp_LE res = FPCompareGE(element2, element1, FPCR[]);
        ElemP[result, e, esize] = if res then '1' else '0';
    else
        ElemP[result, e, esize] = '0';

P[d, PL] = result;
```

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCM<cc> (zero)

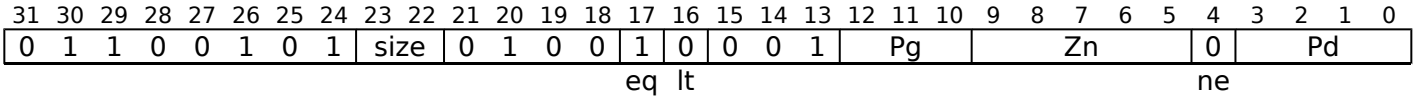
Floating-point compare vector with zero

Compare active floating-point elements in the source vector with zero, and place the boolean results of the specified comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

<cc>	Comparison
EQ	equal
GE	greater than or equal
GT	greater than
LE	less than or equal
LT	less than
NE	not equal
UO	unordered

It has encodings from 6 classes: Equal , Greater than , Greater than or equal , Less than , Less than or equal and Not equal

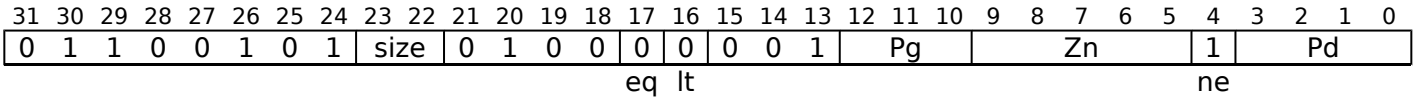
Equal



FCMEQ <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #0.0

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_EQ;
```

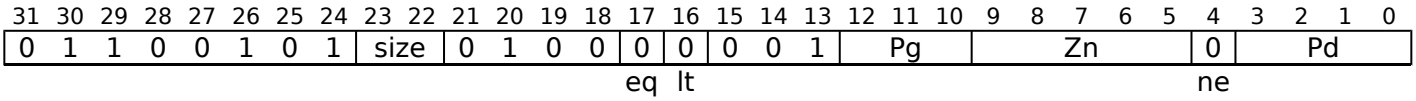
Greater than



FCMGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #0.0

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_GT;
```

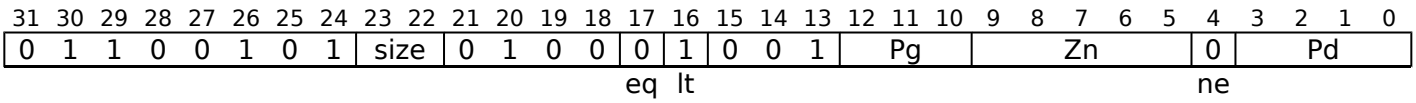
Greater than or equal



FCMGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #0.0

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_GE;
```

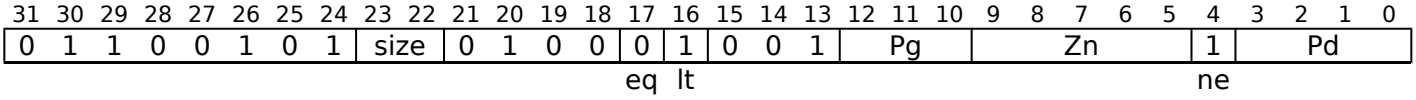
Less than



FCMLT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #0.0

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_LT;
```

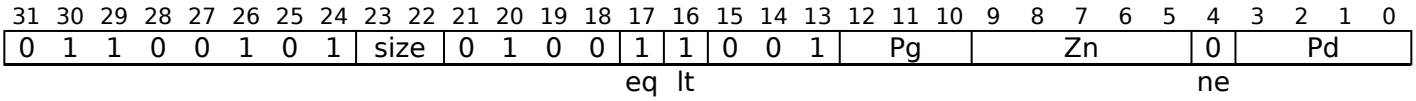
Less than or equal



FCMLE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #0.0

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_LE;
```

Not equal



FCMNE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, #0.0

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Pd);
SVECmp op = Cmp_NE;
```

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(PL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        boolean res;
        case op of
            when Cmp_EQ res = FPCompareEQ(element, 0<esize-1:0>, FPCR[]);
            when Cmp_GE res = FPCompareGE(element, 0<esize-1:0>, FPCR[]);
            when Cmp_GT res = FPCompareGT(element, 0<esize-1:0>, FPCR[]);
            when Cmp_NE res = FPCompareNE(element, 0<esize-1:0>, FPCR[]);
            when Cmp_LT res = FPCompareGT(0<esize-1:0>, element, FPCR[]);
            when Cmp_LE res = FPCompareGE(0<esize-1:0>, element, FPCR[]);
        ElemP[result, e, esize] = if res then '1' else '0';
    else
        ElemP[result, e, esize] = '0';

P[d, PL] = result;
```

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register written by this instruction might be significantly delayed.

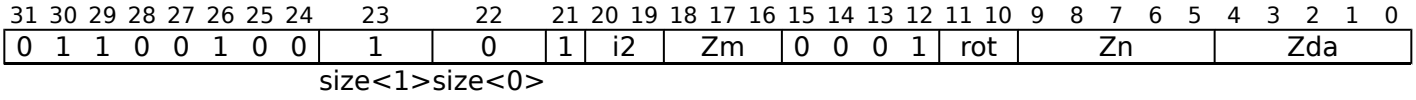
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCMLA (indexed)

Floating-point complex multiply-add by indexed values with rotate

Multiply the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the floating-point complex numbers in each 128-bit segment of the first source vector by the specified complex number in the corresponding the second source vector segment rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation.
Then destructively add the products to the corresponding components of the complex numbers in the addend and destination vector, without intermediate rounding.
These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.
Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.
The complex numbers within the second source vector are specified using an immediate index which selects the same complex number position within each 128-bit vector segment. The index range is from 0 to one less than the number of complex numbers per 128-bit segment, encoded in 1 to 2 bits depending on the size of the complex number. This instruction is unpredicated.
It has encodings from 2 classes: Half-precision and Single-precision

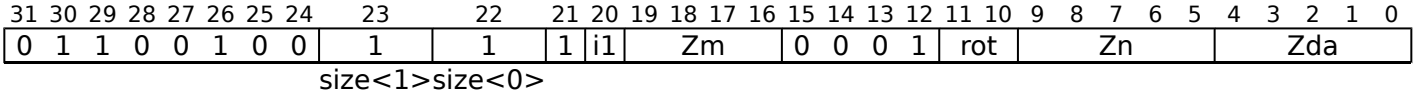
Half-precision



FCMLA <Zda>.H, <Zn>.H, <Zm>.H[<imm>], <const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean neg_i = (rot<1> == '1');
boolean neg_r = (rot<0> != rot<1>);
```

Single-precision



FCMLA <Zda>.S, <Zn>.S, <Zm>.S[<imm>], <const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean neg_i = (rot<1> == '1');
boolean neg_r = (rot<0> != rot<1>);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the half-precision variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

For the single-precision variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the half-precision variant: is the index of a Real and Imaginary pair, in the range 0 to 3, encoded in the "i2" field.

For the single-precision variant: is the index of a Real and Imaginary pair, in the range 0 to 1, encoded in the "i1" field.

<const> Is the const specifier, encoded in “rot”:

rot	<const>
00	#0
01	#90
10	#180
11	#270

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (2 * esize);
constant integer pairspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for p = 0 to pairs-1
    segmentbase = p - (p MOD pairspersegment);
    s = segmentbase + index;
    addend_r = Elem[operand3, 2 * p + 0, esize];
    addend_i = Elem[operand3, 2 * p + 1, esize];
    elt1_a   = Elem[operand1, 2 * p + sel_a, esize];
    elt2_a   = Elem[operand2, 2 * s + sel_a, esize];
    elt2_b   = Elem[operand2, 2 * s + sel_b, esize];
    if neg_r then elt2_a = FPNeg(elt2_a);
    if neg_i then elt2_b = FPNeg(elt2_b);
    addend_r = FPMulAdd(addend_r, elt1_a, elt2_a, FPCR[]);
    addend_i = FPMulAdd(addend_i, elt1_a, elt2_b, FPCR[]);
    Elem[result, 2 * p + 0, esize] = addend_r;
    Elem[result, 2 * p + 1, esize] = addend_i;

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCMLA (vectors)

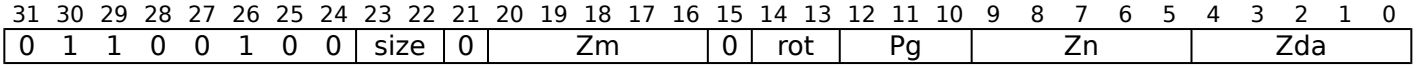
Floating-point complex multiply-add with rotate (predicated)

Multiply the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the floating-point complex numbers in the first source vector by the corresponding complex number in the second source vector rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation.

Then destructively add the products to the corresponding components of the complex numbers in the addend and destination vector, without intermediate rounding.

These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element. Inactive elements in the destination vector register remain unmodified.



FCMLA <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>, <const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean neg_i = (rot<1> == '1');
boolean neg_r = (rot<0> != rot<1>);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<const> Is the const specifier, encoded in “rot”:

rot	<const>
00	#0
01	#90
10	#180
11	#270

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (2 * esize);
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for p = 0 to pairs-1
    addend_r = Elem[operand3, 2 * p + 0, esize];
    addend_i = Elem[operand3, 2 * p + 1, esize];
    if ElemP[mask, 2 * p + 0, esize] == '1' then
        bits(esize) elt1_a = Elem[operand1, 2 * p + sel_a, esize];
        bits(esize) elt2_a = Elem[operand2, 2 * p + sel_a, esize];
        if neg_r then elt2_a = FPNeg(elt2_a);
        addend_r = FPMulAdd(addend_r, elt1_a, elt2_a, FPCR[]);
    if ElemP[mask, 2 * p + 1, esize] == '1' then
        bits(esize) elt1_a = Elem[operand1, 2 * p + sel_a, esize];
        bits(esize) elt2_b = Elem[operand2, 2 * p + sel_b, esize];
        if neg_i then elt2_b = FPNeg(elt2_b);
        addend_i = FPMulAdd(addend_i, elt1_a, elt2_b, FPCR[]);
    Elem[result, 2 * p + 0, esize] = addend_r;
    Elem[result, 2 * p + 1, esize] = addend_i;

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

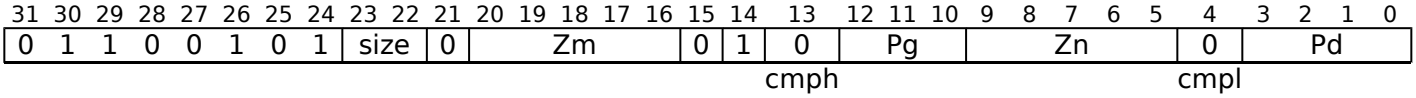
FCMLE (vectors)

Floating-point compare less than or equal to vector

Compare active floating-point elements in the first source vector being less than or equal to corresponding elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is a pseudo-instruction of FCM<cc> (vectors). This means:

- The encodings in this description are named to match the encodings of FCM<cc> (vectors).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of FCM<cc> (vectors) gives the operational pseudocode for this instruction.



FCMLE <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>

is equivalent to

FCMGE <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of FCM<cc> (vectors) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

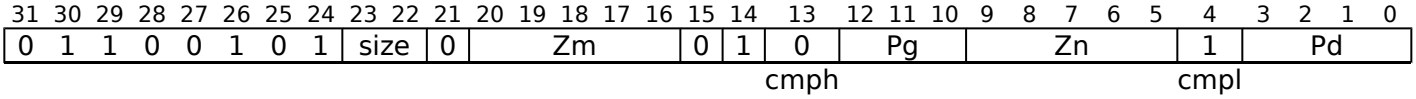
FCMLT (vectors)

Floating-point compare less than vector

Compare active floating-point elements in the first source vector being less than corresponding elements in the second source vector, and place the boolean results of the comparison in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is a pseudo-instruction of FCM<cc> (vectors). This means:

- The encodings in this description are named to match the encodings of FCM<cc> (vectors).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of FCM<cc> (vectors) gives the operational pseudocode for this instruction.



FCMLT <Pd>.<T>, <Pg>/Z, <Zm>.<T>, <Zn>.<T>

is equivalent to

FCMGT <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

Operation

The description of FCM<cc> (vectors) gives the operational pseudocode for this instruction.

Operational information

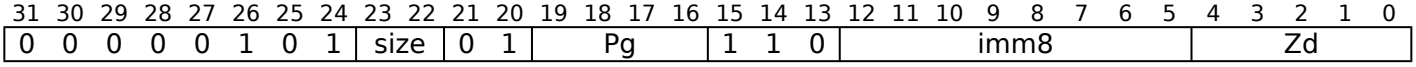
If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCPY

Copy 8-bit floating-point immediate to vector elements (predicated)

Copy a floating-point immediate into each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.
This instruction is used by the alias FMOV (immediate, predicated).



FCPY <Zd>.<T>, <Pg>/M, #<const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer d = UInt(Zd);
bits(esize) imm = VFPExpandImm(imm8, esize);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<const> Is a floating-point immediate value expressable as ±n÷16×2^r, where n and r are integers such that 16 ≤ n ≤ 31 and -3 ≤ r ≤ 4, i.e. a normalized binary floating-point encoding with 1 sign bit, 3-bit exponent, and 4-bit fractional part, encoded in the "imm8" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = imm;

Z[d, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

FCVT

Floating-point convert precision (predicated)

Convert the size and precision of each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.
Since the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are zero-extended to fill each destination element.
It has encodings from 6 classes: Half-precision to single-precision , Half-precision to double-precision , Single-precision to half-precision , Single-precision to double-precision , Double-precision to half-precision and Double-precision to single-precision

Half-precision to single-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	1	0	0	0	1	0	0	1	1	0	1	Pg			Zn					Zd				

FCVT <Zd>.S, <Pg>/M, <Zn>.H

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 16;
constant integer d_esize = 32;
```

Half-precision to double-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	1	1	0	0	1	0	0	1	1	0	1	Pg			Zn					Zd				

FCVT <Zd>.D, <Pg>/M, <Zn>.H

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 16;
constant integer d_esize = 64;
```

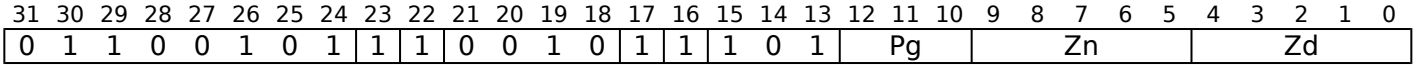
Single-precision to half-precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	1	0	0	0	1	0	0	0	1	0	1	Pg			Zn					Zd				

FCVT <Zd>.H, <Pg>/M, <Zn>.S

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 32;
constant integer d_esize = 16;
```

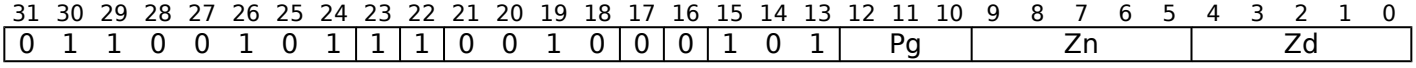
Single-precision to double-precision



FCVT <Zd>.D, <Pg>/M, <Zn>.S

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 32;
constant integer d_esize = 64;
```

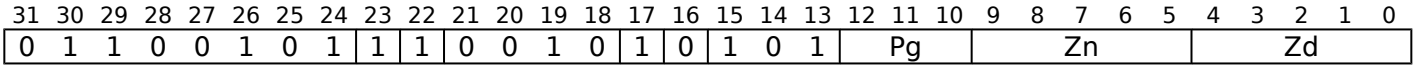
Double-precision to half-precision



FCVT <Zd>.H, <Pg>/M, <Zn>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 64;
constant integer d_esize = 16;
```

Double-precision to single-precision



FCVT <Zd>.S, <Pg>/M, <Zn>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 64;
constant integer d_esize = 32;
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        bits(d_esize) res = FPConvertSVE(element<s_esize-1:0>, FPCR[], d_esize);
        Elem[result, e, esize] = ZeroExtend(res, esize);

Z[d, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

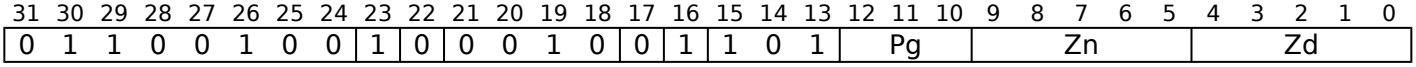
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCVTLT

Floating-point up convert long (top, predicated)

Convert odd-numbered floating-point elements from the source vector to the next higher precision, and place the results in the active overlapping double-width elements of the destination vector. Inactive elements in the destination vector register remain unmodified.
It has encodings from 2 classes: Half-precision to single-precision and Single-precision to double-precision

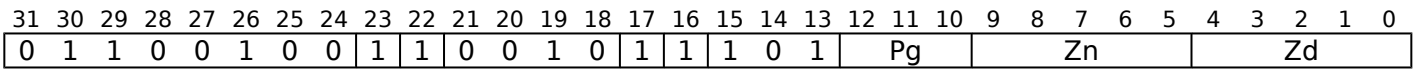
Half-precision to single-precision



FCVTLT <Zd>.S, <Pg>/M, <Zn>.H

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Single-precision to double-precision



FCVTLT <Zd>.D, <Pg>/M, <Zn>.S

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

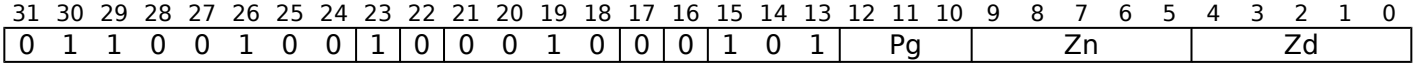
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize DIV 2) element = Elem[operand, 2*e + 1, esize DIV 2];
        Elem[result, e, esize] = FPConvertSVE(element, FPCR[], esize);

Z[d, VL] = result;
```

FCVTNT

Floating-point down convert and narrow (top, predicated)

Convert active floating-point elements from the source vector to the next lower precision, and place the results in the odd-numbered half-width elements of the destination vector, leaving the even-numbered elements unchanged. Inactive elements in the destination vector register remain unmodified.
It has encodings from 2 classes: Single-precision to half-precision and Double-precision to single-precision

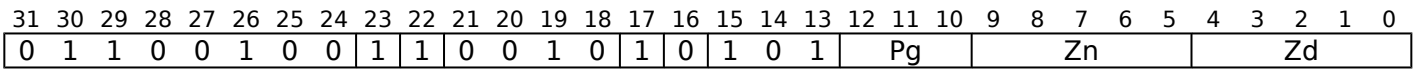
Single-precision to half-precision



FCVTNT <Zd>.H, <Pg>/M, <Zn>.S

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Double-precision to single-precision



FCVTNT <Zd>.S, <Pg>/M, <Zn>.D

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.

Operation

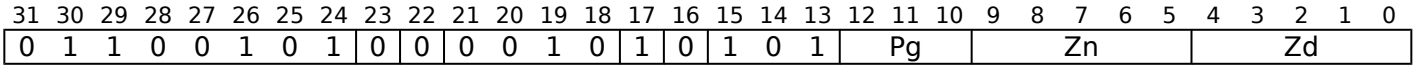
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, 2*e + 1, esize DIV 2] = FPConvertSVE(element, FPCR[], esize DIV 2);

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCVTX

Floating-point down convert, rounding to odd (predicated)

Convert active double-precision floating-point elements from the source vector to single-precision, rounding to Odd, and place the results in the even-numbered 32-bit elements of the destination vector, while setting the odd-numbered elements to zero. Inactive elements in the destination vector register remain unmodified.
Rounding to Odd (aka Von Neumann rounding) permits a two-step conversion from double-precision to half-precision without incurring intermediate rounding errors.



FCVTX <Zd>.S, <Pg>/M, <Zn>.D

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 64;
constant integer d_esize = 32;
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        bits(d_esize) res = FPConvertSVE(element<s_esize-1:0>, FPCR[], FPRounding_ODD, d_esize);
        Elem[result, e, esize] = ZeroExtend(res, esize);

Z[d, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

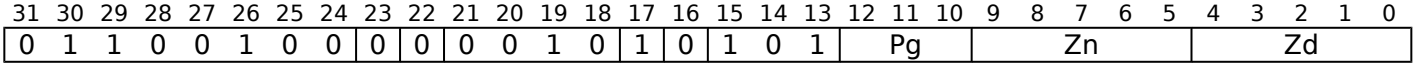
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCVTXNT

Floating-point down convert, rounding to odd (top, predicated)

Convert active double-precision floating-point elements from the source vector to single-precision, rounding to Odd, and place the results in the odd-numbered 32-bit elements of the destination vector, leaving the even-numbered elements unchanged. Inactive elements in the destination vector register remain unmodified.
Rounding to Odd (aka Von Neumann rounding) permits a two-step conversion from double-precision to half-precision without incurring intermediate rounding errors.



FCVTXNT <Zd>.S, <Pg>/M, <Zn>.D

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, 2*e + 1, esize DIV 2] = FPConvertSVE(element, FPCR[], FPRounding_ODD, esize DIV 2);

Z[d, VL] = result;
```

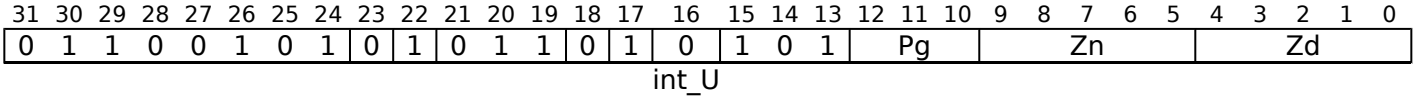
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCVTZS

Floating-point convert to signed integer, rounding toward zero (predicated)

Convert to the signed integer nearer to zero from each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.
If the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are sign-extended to fill each destination element.
It has encodings from 7 classes: Half-precision to 16-bit , Half-precision to 32-bit , Half-precision to 64-bit , Single-precision to 32-bit , Single-precision to 64-bit , Double-precision to 32-bit and Double-precision to 64-bit

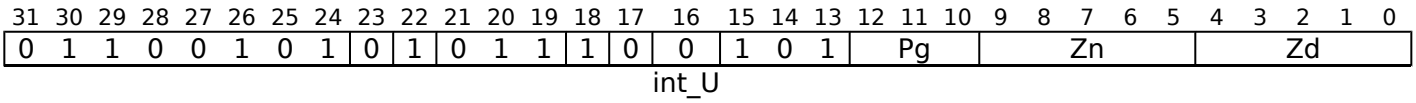
Half-precision to 16-bit



FCVTZS <Zd>.H, <Pg>/M, <Zn>.H

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 16;
constant integer d_esize = 16;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;
```

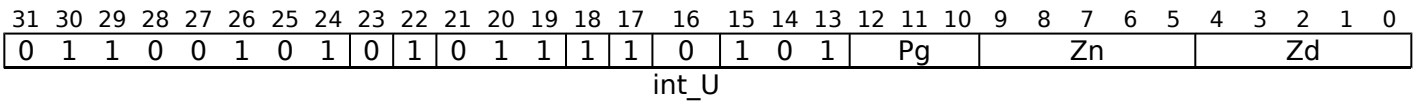
Half-precision to 32-bit



FCVTZS <Zd>.S, <Pg>/M, <Zn>.H

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 16;
constant integer d_esize = 32;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;
```

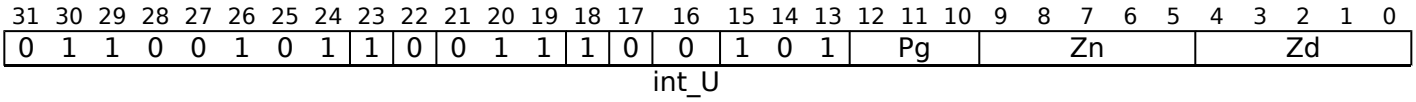
Half-precision to 64-bit



FCVTZS <Zd>.D, <Pg>/M, <Zn>.H

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 16;
constant integer d_esize = 64;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;
```

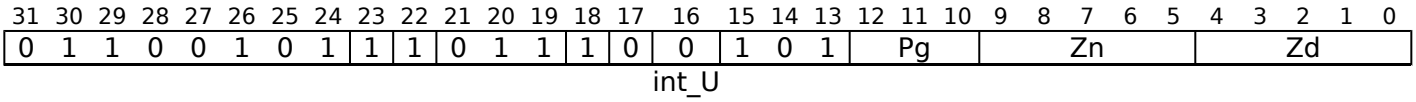
Single-precision to 32-bit



FCVTZS <Zd>.S, <Pg>/M, <Zn>.S

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 32;
constant integer d_esize = 32;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;
```

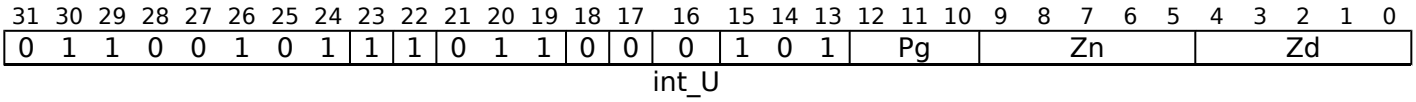
Single-precision to 64-bit



FCVTZS <Zd>.D, <Pg>/M, <Zn>.S

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 32;
constant integer d_esize = 64;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;
```

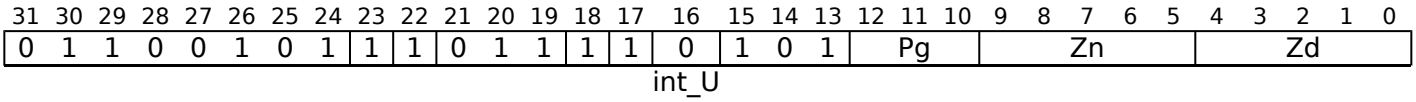
Double-precision to 32-bit



FCVTZS <Zd>.S, <Pg>/M, <Zn>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 64;
constant integer d_esize = 32;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;
```

Double-precision to 64-bit



FCVTZS <Zd>.D, <Pg>/M, <Zn>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 64;
constant integer d_esize = 64;
boolean unsigned = FALSE;
FPRounding rounding = FPRounding_ZERO;
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand  = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        bits(d_esize) res = FPToFixed(element<s_esize-1:0>, 0, unsigned, FPCR[], rounding, d_esize);
        Elem[result, e, esize] = Extend(res, esize, unsigned);

Z[d, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

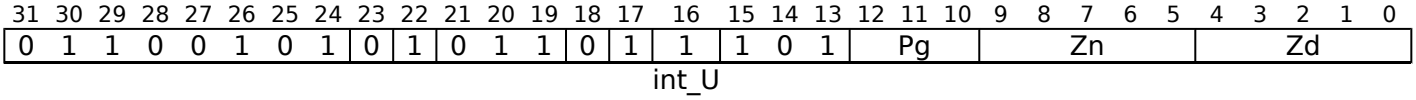
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FCVTZU

Floating-point convert to unsigned integer, rounding toward zero (predicated)

Convert to the unsigned integer nearer to zero from each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.
If the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are zero-extended to fill each destination element.
It has encodings from 7 classes: Half-precision to 16-bit , Half-precision to 32-bit , Half-precision to 64-bit , Single-precision to 32-bit , Single-precision to 64-bit , Double-precision to 32-bit and Double-precision to 64-bit

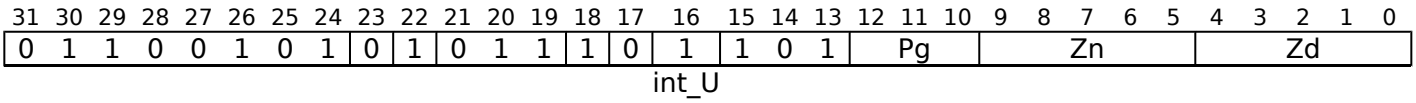
Half-precision to 16-bit



FCVTZU <Zd>.H, <Pg>/M, <Zn>.H

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 16;
constant integer d_esize = 16;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;
```

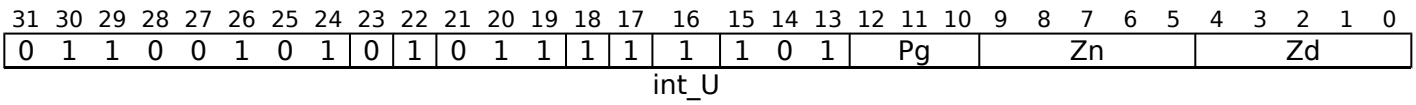
Half-precision to 32-bit



FCVTZU <Zd>.S, <Pg>/M, <Zn>.H

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 16;
constant integer d_esize = 32;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;
```

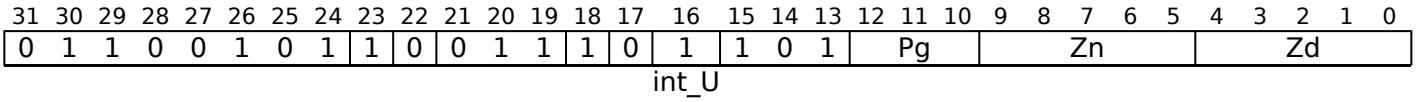
Half-precision to 64-bit



FCVTZU <Zd>.D, <Pg>/M, <Zn>.H

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 16;
constant integer d_esize = 64;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;
```

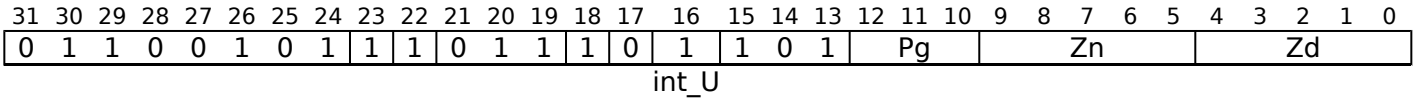
Single-precision to 32-bit



FCVTZU <Zd>.S, <Pg>/M, <Zn>.S

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 32;
constant integer d_esize = 32;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;
```

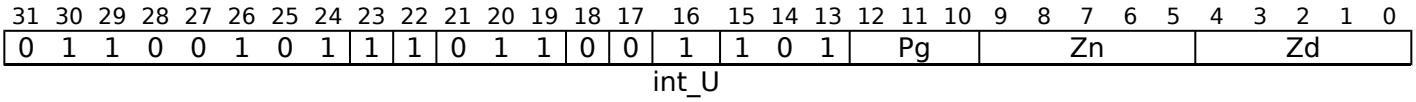
Single-precision to 64-bit



FCVTZU <Zd>.D, <Pg>/M, <Zn>.S

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 32;
constant integer d_esize = 64;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;
```

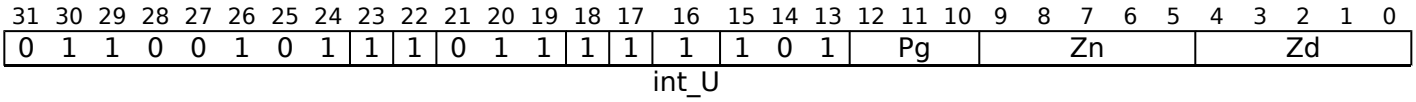
Double-precision to 32-bit



FCVTZU <Zd>.S, <Pg>/M, <Zn>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 64;
constant integer d_esize = 32;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;
```

Double-precision to 64-bit



FCVTZU <Zd>.D, <Pg>/M, <Zn>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 64;
constant integer d_esize = 64;
boolean unsigned = TRUE;
FPRounding rounding = FPRounding_ZERO;
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand  = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        bits(d_esize) res = FPToFixed(element<s_esize-1:0>, 0, unsigned, FPCR[], rounding, d_esize);
        Elem[result, e, esize] = Extend(res, esize, unsigned);

Z[d, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.

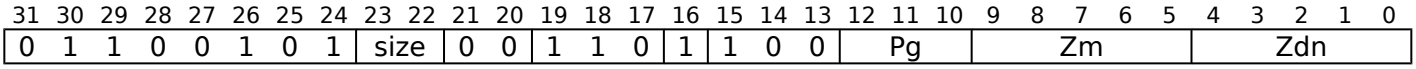
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FDIV

Floating-point divide by vector (predicated)

Divide active floating-point elements of the first source vector by corresponding floating-point elements of the second source vector and destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



FDIV <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = FPDiv(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

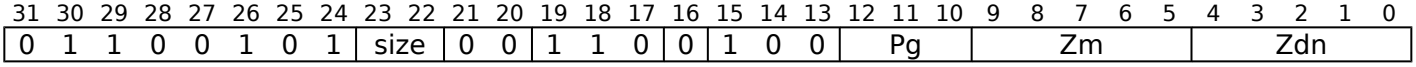
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FDIVR

Floating-point reversed divide by vector (predicated)

Reversed divide active floating-point elements of the second source vector by corresponding floating-point elements of the first source vector and destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



FDIVR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = FPDiv(element2, element1, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

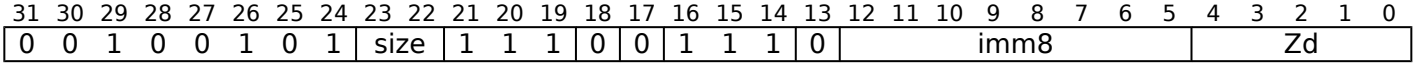
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FDUP

Broadcast 8-bit floating-point immediate to vector elements (unpredicated)

Unconditionally broadcast the floating-point immediate into each element of the destination vector. This instruction is unpredicated.
This instruction is used by the alias FMOV (immediate, unpredicated).



FDUP <Zd>.<T>, #<const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer d = UInt(Zd);
bits(esize) imm = VFPExpandImm(imm8, esize);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<const> Is a floating-point immediate value expressable as ±n÷16×2^r, where n and r are integers such that 16 ≤ n ≤ 31 and -3 ≤ r ≤ 4, i.e. a normalized binary floating-point encoding with 1 sign bit, 3-bit exponent, and 4-bit fractional part, encoded in the "imm8" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) result;

for e = 0 to elements-1
    Elem[result, e, esize] = imm;

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FEXPA

Floating-point exponential accelerator

The FEXPA instruction accelerates the polynomial series calculation of the EXP(X) function.

The double-precision variant copies the low 52 bits of an entry from a hard-wired table of 64-bit coefficients, indexed by the low 6 bits of each element of the source vector, and prepends to that the next 11 bits of the source element (src<16:6>), setting the sign bit to zero.

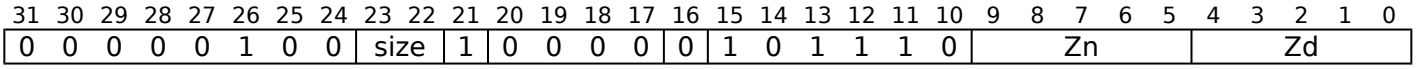
The single-precision variant copies the low 23 bits of an entry from hard-wired table of 32-bit coefficients, indexed by the low 6 bits of each element of the source vector, and prepends to that the next 8 bits of the source element (src<13:6>), setting the sign bit to zero.

The half-precision variant copies the low 10 bits of an entry from hard-wired table of 16-bit coefficients, indexed by the low 5 bits of each element of the source vector, and prepends to that the next 5 bits of the source element (src<9:5>), setting the sign bit to zero.

A coefficient table entry with index M holds the floating-point value $2^{(m/64)}$, or for the half-precision variant $2^{(m/32)}$.

This instruction is unpredicated.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.



FEXPA <Zd>.<T>, <Zn>.<T>

```
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand  = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPExpA(element);

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

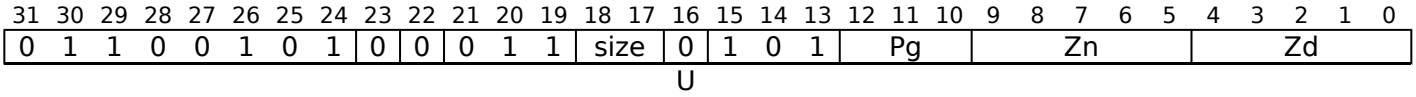
FLOGB

Floating-point base 2 logarithm as integer

This instruction returns the signed integer base 2 logarithm of each floating-point input element |X| after normalization.
This is the unbiased exponent of X used in the representation of the floating-point value, such that, for positive X, X = significand × $2^{exponent}$.
The integer results are placed in elements of the destination vector which have the same width (ESIZE) as the floating-point input elements:

- If X is normal, the result is the base 2 logarithm of X.
- If X is subnormal, the result corresponds to the normalized representation.
- If X is infinite, the result is $2^{(esize-1)}$-1.
- If X is ±0.0 or NaN, the result is $-2^{(esize-1)}$.

Inactive elements in the destination vector register remain unmodified.



FLOGB <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = FPLogB(element, FPCR[]);

Z[d, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

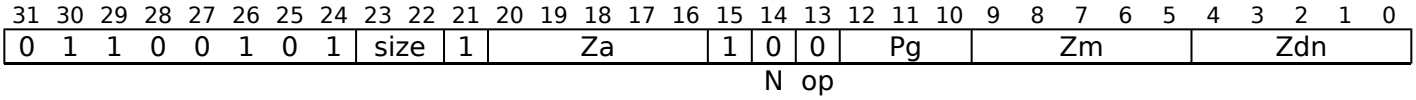
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMAD

Floating-point fused multiply-add vectors (predicated), writing multiplicand [Zdn = Za + Zdn * Zm]

Multiply the corresponding active floating-point elements of the first and second source vectors and add to elements of the third (addend) vector without intermediate rounding. Destructively place the results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.



FMAD <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Za>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
integer a = UInt(Za);
boolean op1_neg = FALSE;
boolean op3_neg = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Za> Is the name of the third source scalable vector register, encoded in the "Za" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = if AnyActiveElement(mask, esize) then Z[a, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        bits(esize) element3 = Elem[operand3, e, esize];

        if op1_neg then element1 = FPNeg(element1);
        if op3_neg then element3 = FPNeg(element3);
        Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

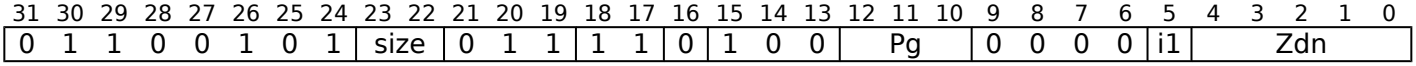
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMAX (immediate)

Floating-point maximum with immediate (predicated)

Determine the maximum of an immediate and each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.0 or +1.0 only. If the element value is NaN then the result is NaN. Inactive elements in the destination vector register remain unmodified.



FMAX <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then Zeros(esize) else FPOne('0', esize);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the floating-point immediate value, encoded in “i1”:

i1	<const>
0	#0.0
1	#1.0

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = FPMax(element1, imm, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

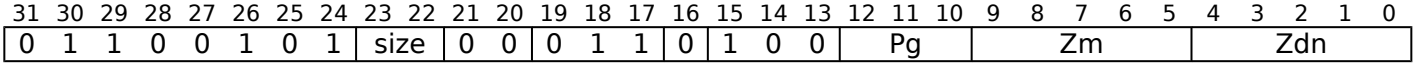
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMAX (vectors)

Floating-point maximum (predicated)

Determine the maximum of active floating-point elements of the second source vector and corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. If either element value is NaN then the result is NaN. Inactive elements in the destination vector register remain unmodified.



FMAX <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = FPMax(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

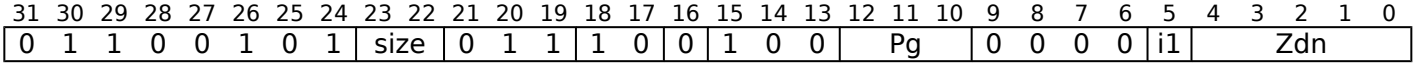
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMAXNM (immediate)

Floating-point maximum number with immediate (predicated)

Determine the maximum number value of an immediate and each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.0 or +1.0 only. If the element value is a quiet NaN, then the result is the immediate. Inactive elements in the destination vector register remain unmodified.



FMAXNM <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then Zeros(esize) else FPOne('0', esize);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the floating-point immediate value, encoded in “i1”:

i1	<const>
0	#0.0
1	#1.0

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = FPMaxNum(element1, imm, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

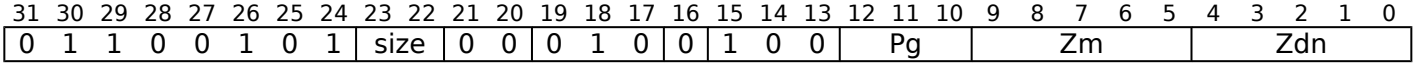
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMAXNM (vectors)

Floating-point maximum number (predicated)

Determine the maximum number value of active floating-point elements of the second source vector and corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. If one element value is numeric and the other is a quiet NaN, then the result is the numeric value. Inactive elements in the destination vector register remain unmodified.



FMAXNM <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

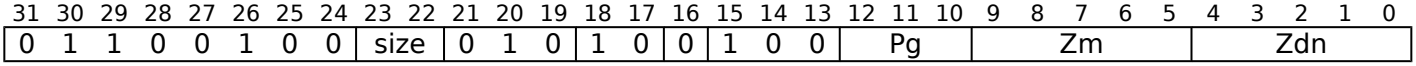
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMAXNMP

Floating-point maximum number pairwise

Compute the maximum value of each pair of adjacent floating-point elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector. NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result is the numerical value.



FMAXNMP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result = Z[dn, VL];
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        if IsEven(e) then
            element1 = Elem[operand1, e + 0, esize];
            element2 = Elem[operand1, e + 1, esize];
        else
            element1 = Elem[operand2, e - 1, esize];
            element2 = Elem[operand2, e + 0, esize];
        Elem[result, e, esize] = FPMaxNum(element1, element2, FPCR[]);

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

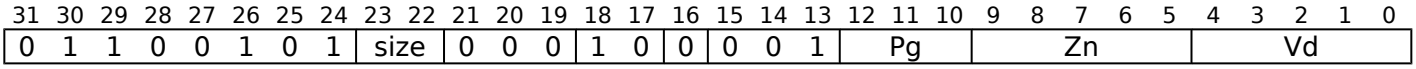
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMAXNMV

Floating-point maximum number recursive reduction to scalar

Floating-point maximum number horizontally over all lanes of a vector using a recursive pairwise reduction, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the default NaN.



FMAXNMV <V><d>, <Pg>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	D

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

Operation

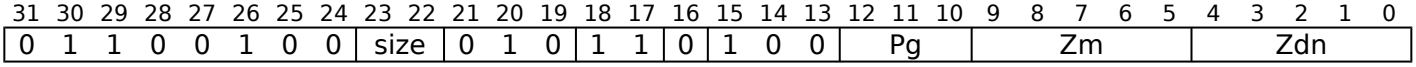
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(esize) identity = FPDefaultNaN(esize);

V[d, esize] = ReducePredicated(ReduceOp_FMAXNUM, operand, mask, identity);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMAXP

Floating-point maximum pairwise

Compute the maximum value of each pair of adjacent floating-point elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.



FMAXP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result = Z[dn, VL];
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        if IsEven(e) then
            element1 = Elem[operand1, e + 0, esize];
            element2 = Elem[operand1, e + 1, esize];
        else
            element1 = Elem[operand2, e - 1, esize];
            element2 = Elem[operand2, e + 0, esize];
        Elem[result, e, esize] = FPMax(element1, element2, FPCR[]);

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

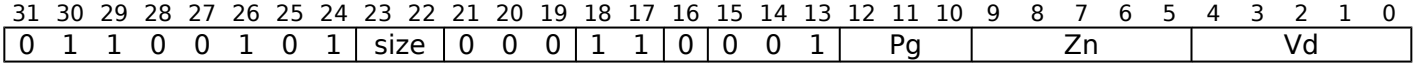
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMAXV

Floating-point maximum recursive reduction to scalar

Floating-point maximum horizontally over all lanes of a vector using a recursive pairwise reduction, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as -Infinity.



FMAXV <V><d>, <Pg>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	D

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

Operation

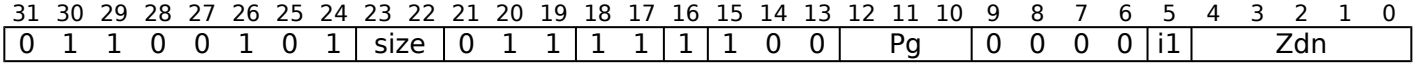
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(esize) identity = FPInfinity('1', esize);

V[d, esize] = ReducePredicated(ReduceOp_FMAX, operand, mask, identity);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMIN (immediate)

Floating-point minimum with immediate (predicated)

Determine the minimum of an immediate and each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.0 or +1.0 only. If the element value is NaN then the result is NaN. Inactive elements in the destination vector register remain unmodified.



FMIN <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then Zeros(esize) else FPOne('0', esize);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the floating-point immediate value, encoded in “i1”:

i1	<const>
0	#0.0
1	#1.0

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = FPMin(element1, imm, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

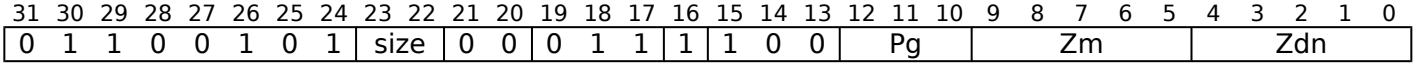
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMIN (vectors)

Floating-point minimum (predicated)

Determine the minimum of active floating-point elements of the second source vector and corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. If either element value is NaN then the result is NaN. Inactive elements in the destination vector register remain unmodified.



FMIN <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = FPMin(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

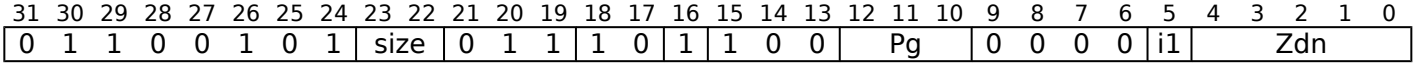
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMINNM (immediate)

Floating-point minimum number with immediate (predicated)

Determine the minimum number value of an immediate and each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.0 or +1.0 only. If the element value is a quiet NaN, then the result is the immediate. Inactive elements in the destination vector register remain unmodified.



FMINNM <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then Zeros(esize) else FPOne('0', esize);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the floating-point immediate value, encoded in “i1”:

i1	<const>
0	#0.0
1	#1.0

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = FPMinNum(element1, imm, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

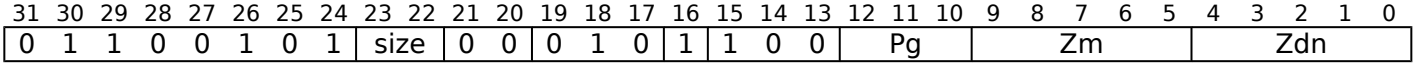
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMINNM (vectors)

Floating-point minimum number (predicated)

Determine the minimum number value of active floating-point elements of the second source vector and corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. If one element value is numeric and the other is a quiet NaN, then the result is the numeric value. Inactive elements in the destination vector register remain unmodified.



FMINNM <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = FPMinNum(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

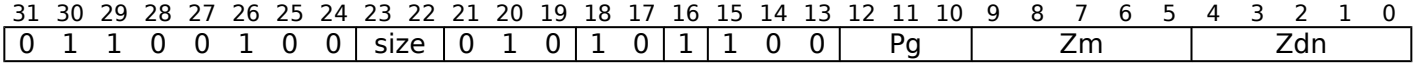
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMINNMP

Floating-point minimum number pairwise

Compute the minimum value of each pair of adjacent floating-point elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector. NaNs are handled according to the IEEE 754-2008 standard. If one vector element is numeric and the other is a quiet NaN, the result is the numerical value.



FMINNMP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result = Z[dn, VL];
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        if IsEven(e) then
            element1 = Elem[operand1, e + 0, esize];
            element2 = Elem[operand1, e + 1, esize];
        else
            element1 = Elem[operand2, e - 1, esize];
            element2 = Elem[operand2, e + 0, esize];
        Elem[result, e, esize] = FPMinNum(element1, element2, FPCR[]);

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

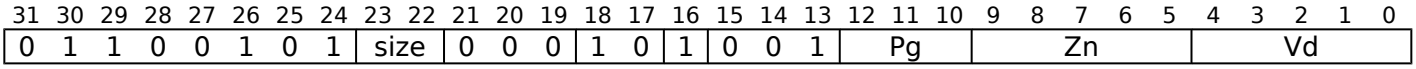
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMINNMV

Floating-point minimum number recursive reduction to scalar

Floating-point minimum number horizontally over all lanes of a vector using a recursive pairwise reduction, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the default NaN.



FMINNMV <V><d>, <Pg>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	D

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

Operation

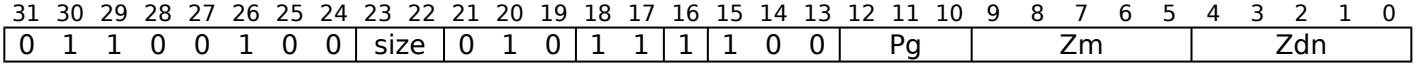
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(esize) identity = FPDefaultNaN(esize);

V[d, esize] = ReducePredicated(ReduceOp_FMINNUM, operand, mask, identity);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMINP

Floating-point minimum pairwise

Compute the minimum value of each pair of adjacent floating-point elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.



FMINP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result = Z[dn, VL];
bits(esize) element1;
bits(esize) element2;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        if IsEven(e) then
            element1 = Elem[operand1, e + 0, esize];
            element2 = Elem[operand1, e + 1, esize];
        else
            element1 = Elem[operand2, e - 1, esize];
            element2 = Elem[operand2, e + 0, esize];
        Elem[result, e, esize] = FPMin(element1, element2, FPCR[]);

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

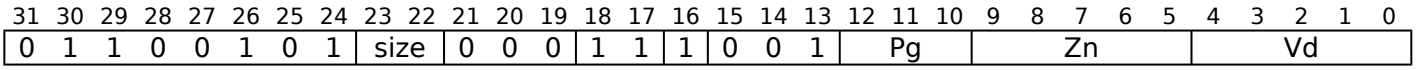
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMINV

Floating-point minimum recursive reduction to scalar

Floating-point minimum horizontally over all lanes of a vector using a recursive pairwise reduction, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as +Infinity.



FMINV <V><d>, <Pg>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	RESERVED
01	H
10	S
11	D

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(esize) identity = FPInfinity('0', esize);

V[d, esize] = ReducePredicated(ReduceOp_FMIN, operand, mask, identity);
```

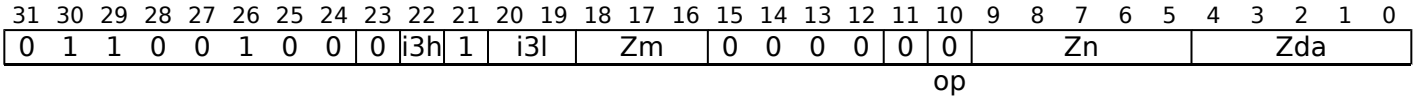
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMLA (indexed)

Floating-point fused multiply-add by indexed elements (Zda = Zda + Zn * Zm[indexed])

Multiply all floating-point elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The products are then destructively added without intermediate rounding to the corresponding elements of the addend and destination vector.
The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element. This instruction is unpredicated.
It has encodings from 3 classes: Half-precision , Single-precision and Double-precision

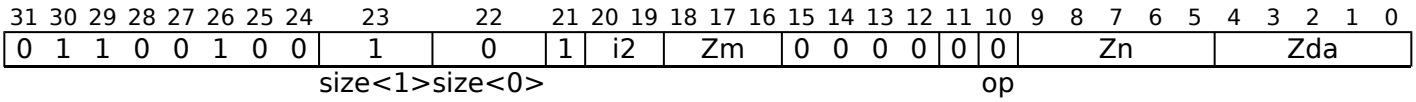
Half-precision



FMLA <Zda>.H, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;
boolean op3_neg = FALSE;
```

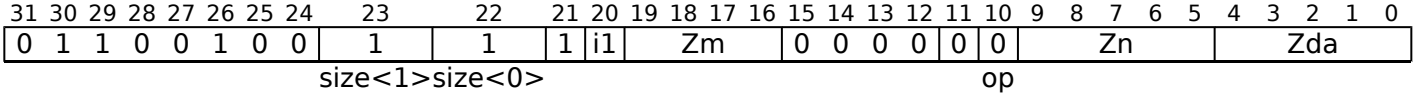
Single-precision



FMLA <Zda>.S, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;
boolean op3_neg = FALSE;
```

Double-precision



FMLA <Zda>.D, <Zn>.D, <Zm>.D[<imm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;
boolean op3_neg = FALSE;
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the half-precision and single-precision variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field. For the double-precision variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the half-precision variant: is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields. For the single-precision variant: is the immediate index, in the range 0 to 3, encoded in the "i2" field. For the double-precision variant: is the immediate index, in the range 0 to 1, encoded in the "i1" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, s, esize];
    bits(esize) element3 = Elem[result, e, esize];
    if op1_neg then element1 = FPNeg(element1);
    if op3_neg then element3 = FPNeg(element3);
    Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

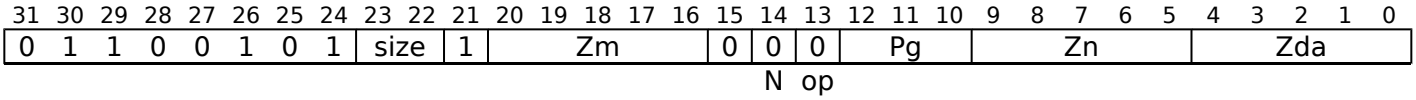
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMLA (vectors)

Floating-point fused multiply-add vectors (predicated), writing addend [Zda = Zda + Zn * Zm]

Multiply the corresponding active floating-point elements of the first and second source vectors and add to elements of the third source (addend) vector without intermediate rounding. Destructively place the results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.



FMLA <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;
boolean op3_neg = FALSE;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        bits(esize) element3 = Elem[operand3, e, esize];

        if op1_neg then element1 = FPNeg(element1);
        if op3_neg then element3 = FPNeg(element3);
        Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = Elem[operand3, e, esize];

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

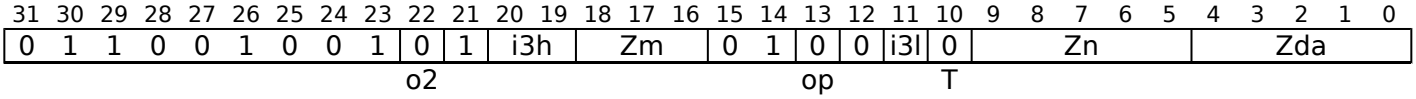
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMLALB (indexed)

Half-precision floating-point multiply-add long to single-precision (bottom, indexed)

This half-precision floating-point multiply-add long instruction widens the even-numbered half-precision elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the first source vector. This instruction is unpredicated.



FMLALB <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i3h:i3l);
boolean op1_neg = FALSE;
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
<imm>	Is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = 2 * segmentbase + index;
    bits(esize DIV 2) element1 = Elem[operand1, 2 * e + 0, esize DIV 2];
    bits(esize DIV 2) element2 = Elem[operand2, s, esize DIV 2];
    bits(esize) element3 = Elem[operand3, e, esize];
    if op1_neg then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

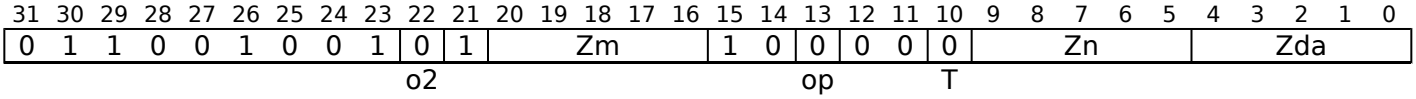
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMLALB (vectors)

Half-precision floating-point multiply-add long to single-precision (bottom)

This half-precision floating-point multiply-add long instruction widens the even-numbered half-precision elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the source vectors. This instruction is unpredicated.



FMLALB <Zda>.S, <Zn>.H, <Zm>.H

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize DIV 2) element1 = Elem[operand1, 2 * e + 0, esize DIV 2];
    bits(esize DIV 2) element2 = Elem[operand2, 2 * e + 0, esize DIV 2];
    bits(esize) element3 = Elem[operand3, e, esize];
    if op1_neg then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

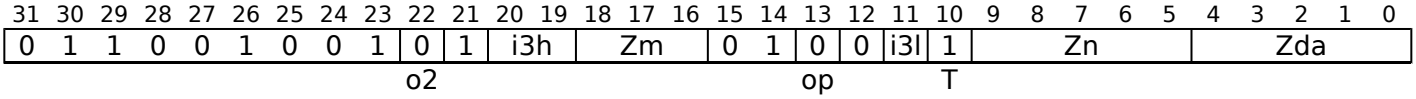
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMLALT (indexed)

Half-precision floating-point multiply-add long to single-precision (top, indexed)

This half-precision floating-point multiply-add long instruction widens the odd-numbered half-precision elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the first source vector. This instruction is unpredicated.



FMLALT <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i3h:i3l);
boolean op1_neg = FALSE;
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
<imm>	Is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = 2 * segmentbase + index;
    bits(esize DIV 2) element1 = Elem[operand1, 2 * e + 1, esize DIV 2];
    bits(esize DIV 2) element2 = Elem[operand2, s, esize DIV 2];
    bits(esize) element3 = Elem[operand3, e, esize];
    if op1_neg then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

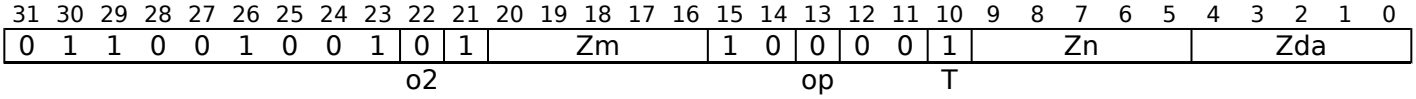
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMLALT (vectors)

Half-precision floating-point multiply-add long to single-precision (top)

This half-precision floating-point multiply-add long instruction widens the odd-numbered half-precision elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and adds these values without intermediate rounding to the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the source vectors. This instruction is unpredicated.



FMLALT <Zda>.S, <Zn>.H, <Zm>.H

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize DIV 2) element1 = Elem[operand1, 2 * e + 1, esize DIV 2];
    bits(esize DIV 2) element2 = Elem[operand2, 2 * e + 1, esize DIV 2];
    bits(esize) element3 = Elem[operand3, e, esize];
    if op1_neg then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

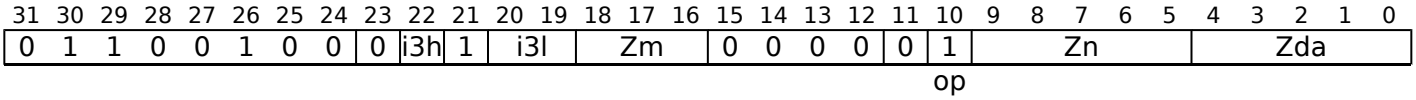
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMLS (indexed)

Floating-point fused multiply-subtract by indexed elements (Zda = Zda + -Zn * Zm[indexed])

Multiply all floating-point elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The products are then destructively subtracted without intermediate rounding from the corresponding elements of the addend and destination vector.
The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element. This instruction is unpredicated.
It has encodings from 3 classes: Half-precision , Single-precision and Double-precision

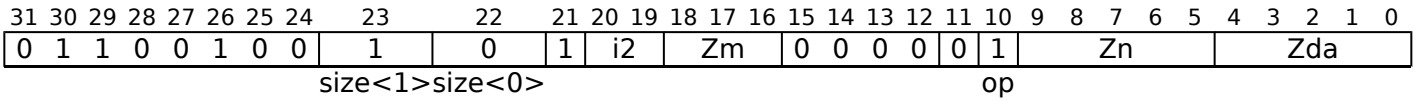
Half-precision



FMLS <Zda>.H, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
boolean op3_neg = FALSE;
```

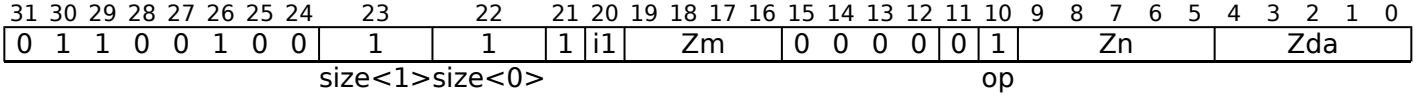
Single-precision



FMLS <Zda>.S, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
boolean op3_neg = FALSE;
```

Double-precision



FMLS <Zda>.D, <Zn>.D, <Zm>.D[<imm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
boolean op3_neg = FALSE;
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the half-precision and single-precision variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field. For the double-precision variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the half-precision variant: is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields. For the single-precision variant: is the immediate index, in the range 0 to 3, encoded in the "i2" field. For the double-precision variant: is the immediate index, in the range 0 to 1, encoded in the "i1" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, s, esize];
    bits(esize) element3 = Elem[result, e, esize];
    if op1_neg then element1 = FPNeg(element1);
    if op3_neg then element3 = FPNeg(element3);
    Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

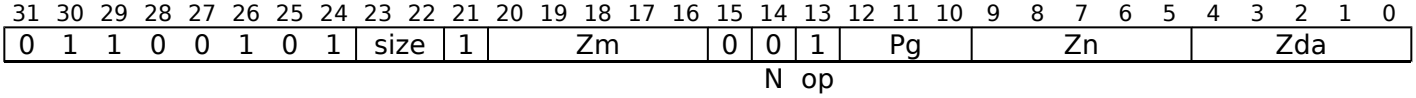
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMLS (vectors)

Floating-point fused multiply-subtract vectors (predicated), writing addend [Zda = Zda + -Zn * Zm]

Multiply the corresponding active floating-point elements of the first and second source vectors and subtract from elements of the third source (addend) vector without intermediate rounding. Destructively place the results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.



FMLS <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
boolean op3_neg = FALSE;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        bits(esize) element3 = Elem[operand3, e, esize];

        if op1_neg then element1 = FPNeg(element1);
        if op3_neg then element3 = FPNeg(element3);
        Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = Elem[operand3, e, esize];

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

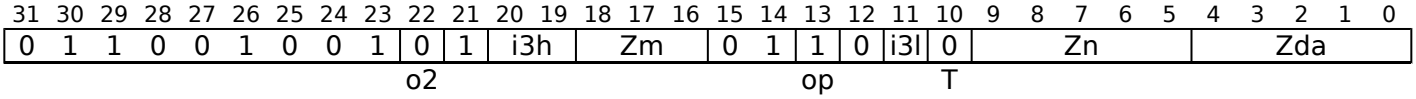
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMLSLB (indexed)

Half-precision floating-point multiply-subtract long from single-precision (bottom, indexed)

This half-precision floating-point multiply-subtract long instruction widens the even-numbered half-precision elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the first source vector. This instruction is unpredicated.



FMLSLB <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i3h:i3l);
boolean op1_neg = TRUE;
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
<imm>	Is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = 2 * segmentbase + index;
    bits(esize DIV 2) element1 = Elem[operand1, 2 * e + 0, esize DIV 2];
    bits(esize DIV 2) element2 = Elem[operand2, s, esize DIV 2];
    bits(esize) element3 = Elem[operand3, e, esize];
    if op1_neg then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

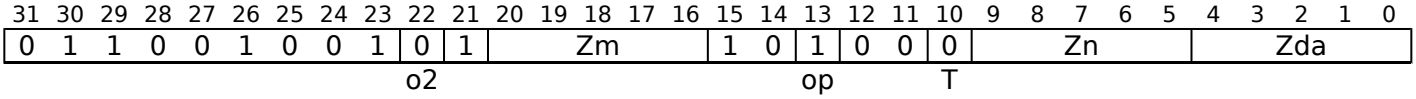
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMLSLB (vectors)

Half-precision floating-point multiply-subtract long from single-precision (bottom)

This half-precision floating-point multiply-subtract long instruction widens the even-numbered half-precision elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the source vectors. This instruction is unpredicated.



FMLSLB <Zda>.S, <Zn>.H, <Zm>.H

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize DIV 2) element1 = Elem[operand1, 2 * e + 0, esize DIV 2];
    bits(esize DIV 2) element2 = Elem[operand2, 2 * e + 0, esize DIV 2];
    bits(esize) element3 = Elem[operand3, e, esize];
    if op1_neg then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

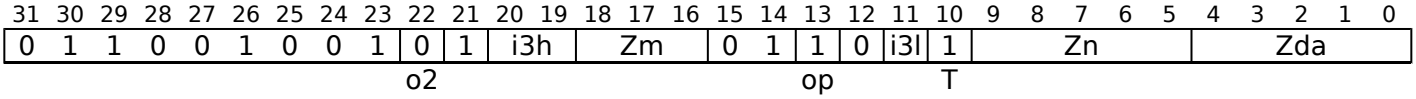
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMLSLT (indexed)

Half-precision floating-point multiply-subtract long from single-precision (top, indexed)

This half-precision floating-point multiply-subtract long instruction widens the odd-numbered half-precision elements in the first source vector and the indexed element from the corresponding 128-bit segment in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the first source vector. This instruction is unpredicated.



FMLSLT <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer index = UInt(i3h:i3l);
boolean op1_neg = TRUE;
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
<imm>	Is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = 2 * segmentbase + index;
    bits(esize DIV 2) element1 = Elem[operand1, 2 * e + 1, esize DIV 2];
    bits(esize DIV 2) element2 = Elem[operand2, s, esize DIV 2];
    bits(esize) element3 = Elem[operand3, e, esize];
    if op1_neg then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

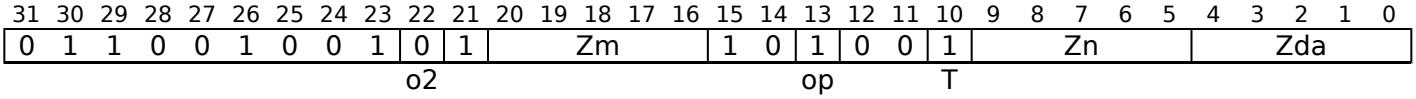
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMLSLT (vectors)

Half-precision floating-point multiply-subtract long from single-precision (top)

This half-precision floating-point multiply-subtract long instruction widens the odd-numbered half-precision elements in the first source vector and the corresponding elements in the second source vector to single-precision format and then destructively multiplies and subtracts these values without intermediate rounding from the single-precision elements of the destination vector that overlap with the corresponding half-precision elements in the source vectors. This instruction is unpredicated.



FMLSLT <Zda>.S, <Zn>.H, <Zm>.H

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize DIV 2) element1 = Elem[operand1, 2 * e + 1, esize DIV 2];
    bits(esize DIV 2) element2 = Elem[operand2, 2 * e + 1, esize DIV 2];
    bits(esize) element3 = Elem[operand3, e, esize];
    if op1_neg then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(element3, element1, element2, FPCR[]);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMMLA

Floating-point matrix multiply-accumulate

The floating-point matrix multiply-accumulate instruction supports single-precision and double-precision data types in a 2×2 matrix contained in segments of 128 or 256 bits, respectively. It multiplies the 2×2 matrix in each segment of the first source vector by the 2×2 matrix in the corresponding segment of the second source vector. The resulting 2×2 matrix product is then destructively added to the matrix accumulator held in the corresponding segment of the addend and destination vector. This is equivalent to performing a 2-way dot product per destination element. This instruction is unpredicated. The single-precision variant is vector length agnostic. The double-precision variant requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits are set to zero.

ID_AA64ZFR0_EL1.F32MM indicates whether the single-precision variant is implemented.

ID_AA64ZFR0_EL1.F64MM indicates whether the double-precision variant is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

It has encodings from 2 classes: 32-bit element and 64-bit element

32-bit element

(FEAT_F32MM)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	1	0	1	Zm					1	1	1	0	0	1	Zn					Zda				

FMMLA <Zda>.S, <Zn>.S, <Zm>.S

```
if !HaveSVE() || !HaveSVEFP32MatMulExt() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

64-bit element

(FEAT_F64MM)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	1	1	1	Zm					1	1	1	0	0	1	Zn					Zda				

FMMLA <Zda>.D, <Zn>.D, <Zm>.D

```
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
constant integer esize = 64;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
if VL < esize * 4 then UNDEFINED;
constant integer segments = VL DIV (4 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result = Zeros(VL);
bits(4*esize) op1, op2;
bits(4*esize) res, addend;

for s = 0 to segments-1
    op1    = Elem[operand1, s, 4*esize];
    op2    = Elem[operand2, s, 4*esize];
    addend = Elem[operand3, s, 4*esize];
    res    = FPMatMulAdd(addend, op1, op2, esize, FPCR[]);
    Elem[result, s, 4*esize] = res;

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

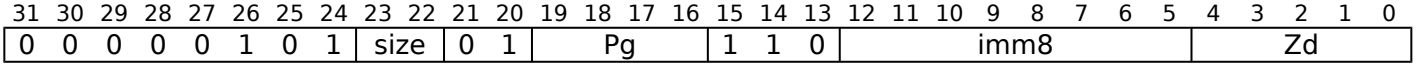
FMOV (immediate, predicated)

Move 8-bit floating-point immediate to vector elements (predicated)

Move a floating-point immediate into each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This is an alias of FCPY. This means:

- The encodings in this description are named to match the encodings of FCPY.
- The description of FCPY gives the operational pseudocode for this instruction.



FMOV <Zd>.<T>, <Pg>/M, #<const>

is equivalent to

FCPY <Zd>.<T>, <Pg>/M, #<const>

and is always the preferred disassembly.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<const> Is a floating-point immediate value expressable as ±n÷16×2^r, where n and r are integers such that 16 ≤ n ≤ 31 and -3 ≤ r ≤ 4, i.e. a normalized binary floating-point encoding with 1 sign bit, 3-bit exponent, and 4-bit fractional part, encoded in the "imm8" field.

Operation

The description of FCPY gives the operational pseudocode for this instruction.

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

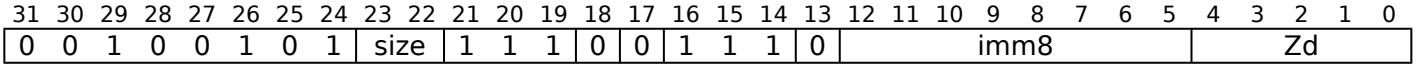
FMOV (immediate, unpredicated)

Move 8-bit floating-point immediate to vector elements (unpredicated)

Unconditionally broadcast the floating-point immediate into each element of the destination vector. This instruction is unpredicated.

This is an alias of FDUP. This means:

- The encodings in this description are named to match the encodings of FDUP.
- The description of FDUP gives the operational pseudocode for this instruction.



FMOV <Zd>.<T>, #<const>

is equivalent to

FDUP <Zd>.<T>, #<const>

and is always the preferred disassembly.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<const> Is a floating-point immediate value expressable as ±n÷16×2^r, where n and r are integers such that 16 ≤ n ≤ 31 and -3 ≤ r ≤ 4, i.e. a normalized binary floating-point encoding with 1 sign bit, 3-bit exponent, and 4-bit fractional part, encoded in the "imm8" field.

Operation

The description of FDUP gives the operational pseudocode for this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

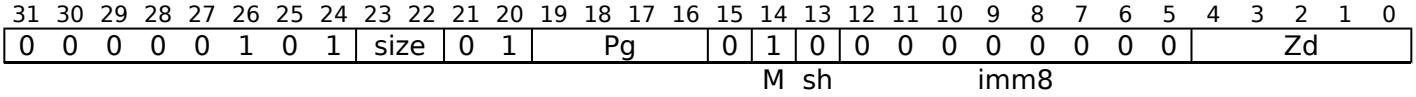
FMOV (zero, predicated)

Move floating-point +0.0 to vector elements (predicated)

Move floating-point constant +0.0 to to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This is a pseudo-instruction of CPY (immediate, merging). This means:

- The encodings in this description are named to match the encodings of CPY (immediate, merging).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of CPY (immediate, merging) gives the operational pseudocode for this instruction.



FMOV <Zd>.<T>, <Pg>/M, #0.0

is equivalent to

CPY <Zd>.<T>, <Pg>/M, #0

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

Operation

The description of CPY (immediate, merging) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

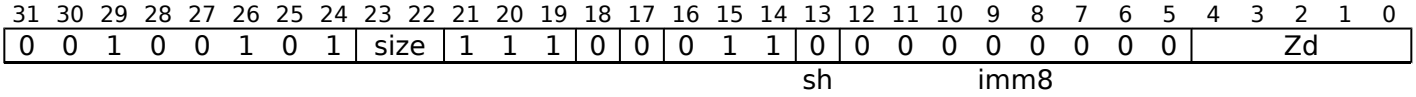
FMOV (zero, unpredicated)

Move floating-point +0.0 to vector elements (unpredicated)

Unconditionally broadcast the floating-point constant +0.0 into each element of the destination vector. This instruction is unpredicated.

This is a pseudo-instruction of DUP (immediate). This means:

- The encodings in this description are named to match the encodings of DUP (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of DUP (immediate) gives the operational pseudocode for this instruction.



FMOV <Zd>.<T>, #0.0

is equivalent to

DUP <Zd>.<T>, #0

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

Operation

The description of DUP (immediate) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

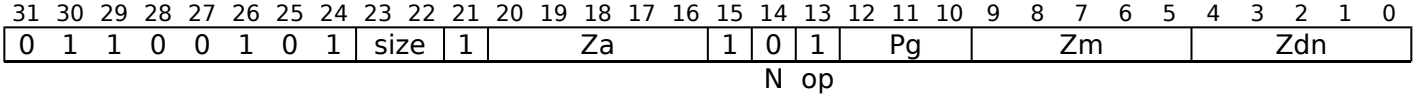
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMSB

Floating-point fused multiply-subtract vectors (predicated), writing multiplicand [Zdn = Za + -Zdn * Zm]

Multiply the corresponding active floating-point elements of the first and second source vectors and subtract from elements of the third (addend) vector without intermediate rounding. Destructively place the results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.



FMSB <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Za>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
integer a = UInt(Za);
boolean op1_neg = TRUE;
boolean op3_neg = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Za> Is the name of the third source scalable vector register, encoded in the "Za" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = if AnyActiveElement(mask, esize) then Z[a, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        bits(esize) element3 = Elem[operand3, e, esize];

        if op1_neg then element1 = FPNeg(element1);
        if op3_neg then element3 = FPNeg(element3);
        Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

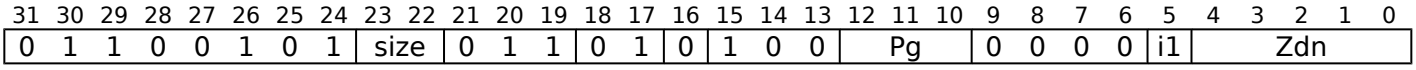
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMUL (immediate)

Floating-point multiply by immediate (predicated)

Multiply by an immediate each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.5 or +2.0 only. Inactive elements in the destination vector register remain unmodified.



FMUL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then FPPointFive('0', esize) else FPTwo('0', esize);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the floating-point immediate value, encoded in “i1”:

i1	<const>
0	#0.5
1	#2.0

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = FPMul(element1, imm, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

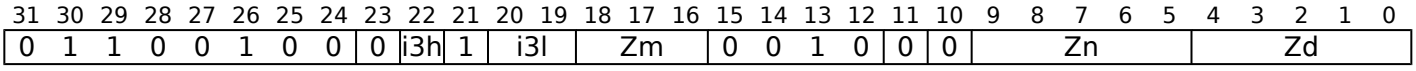
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMUL (indexed)

Floating-point multiply by indexed elements

Multiply all floating-point elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The results are placed in the corresponding elements of the destination vector.
The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element. This instruction is unpredicated.
It has encodings from 3 classes: Half-precision , Single-precision and Double-precision

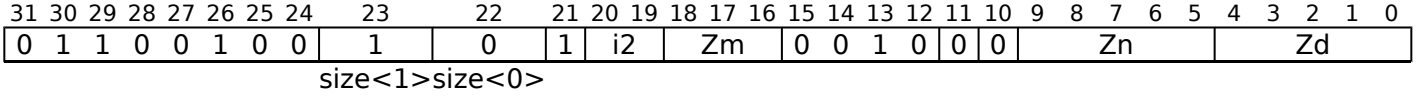
Half-precision



FMUL <Zd>.H, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

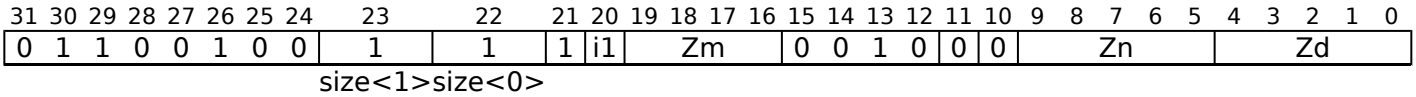
Single-precision



FMUL <Zd>.S, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Double-precision



FMUL <Zd>.D, <Zn>.D, <Zm>.D[<imm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the half-precision and single-precision variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

For the double-precision variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the half-precision variant: is the immediate index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

For the single-precision variant: is the immediate index, in the range 0 to 3, encoded in the "i2" field.

For the double-precision variant: is the immediate index, in the range 0 to 1, encoded in the "i1" field.

Operation

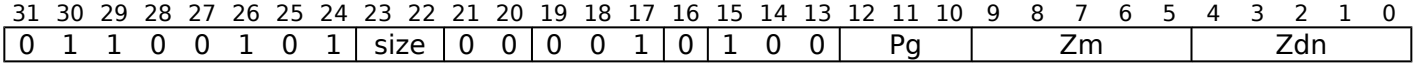
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, s, esize];
    Elem[result, e, esize] = FPMul(element1, element2, FPCR[]);

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMUL (vectors, predicated)

Floating-point multiply vectors (predicated)

Multiply active floating-point elements of the first source vector by corresponding floating-point elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



FMUL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = FPMul(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

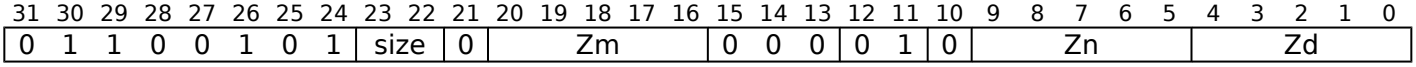
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMUL (vectors, unpredicated)

Floating-point multiply vectors (unpredicated)

Multiply all elements of the first source vector by corresponding floating-point elements of the second source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.



FMUL <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

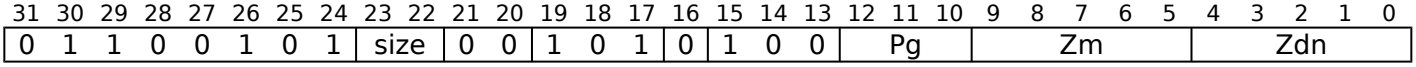
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPMul(element1, element2, FPCR[]);

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMULX

Floating-point multiply-extended vectors (predicated)

Multiply active floating-point elements of the first source vector by corresponding floating-point elements of the second source vector except that ∞×0.0 gives 2.0 instead of NaN, and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.
The instruction can be used with FRECPX to safely convert arbitrary elements in mathematical vector space to UNIT VECTORS or DIRECTION VECTORS with length 1.



FMULX <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = FPMulX(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

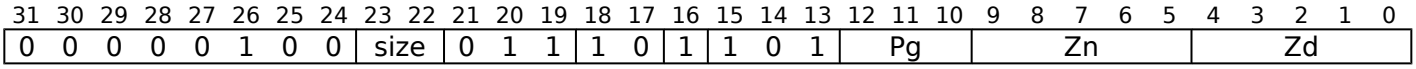
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FNEG

Floating-point negate (predicated)

Negate each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. This inverts the sign bit and cannot signal a floating-point exception. Inactive elements in the destination vector register remain unmodified.



FNEG <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = FPNeg(element);

Z[d, VL] = result;
```

Operational information

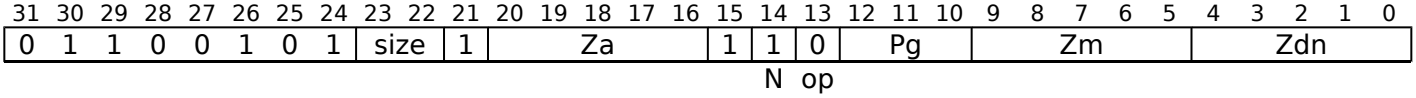
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

FNMAD

Floating-point negated fused multiply-add vectors (predicated), writing multiplicand [Zdn = -Za + -Zdn * Zm]

Multiply the corresponding active floating-point elements of the first and second source vectors and add to elements of the third (addend) vector without intermediate rounding. Destructively place the negated results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.



FNMAD <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Za>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
integer a = UInt(Za);
boolean op1_neg = TRUE;
boolean op3_neg = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Za> Is the name of the third source scalable vector register, encoded in the "Za" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = if AnyActiveElement(mask, esize) then Z[a, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        bits(esize) element3 = Elem[operand3, e, esize];

        if op1_neg then element1 = FPNeg(element1);
        if op3_neg then element3 = FPNeg(element3);
        Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

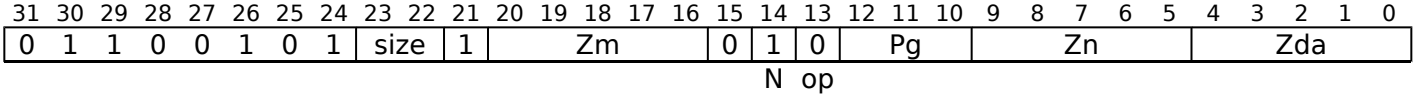
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FNMLA

Floating-point negated fused multiply-add vectors (predicated), writing addend [Zda = -Zda + -Zn * Zm]

Multiply the corresponding active floating-point elements of the first and second source vectors and add to elements of the third source (addend) vector without intermediate rounding. Destructively place the negated results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.



FNMLA <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = TRUE;
boolean op3_neg = TRUE;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        bits(esize) element3 = Elem[operand3, e, esize];

        if op1_neg then element1 = FPNeg(element1);
        if op3_neg then element3 = FPNeg(element3);
        Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = Elem[operand3, e, esize];

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

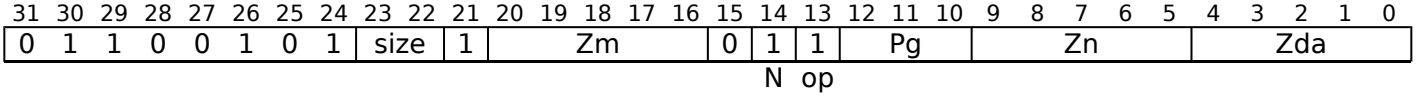
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FNMLS

Floating-point negated fused multiply-subtract vectors (predicated), writing addend [Zda = -Zda + Zn * Zm]

Multiply the corresponding active floating-point elements of the first and second source vectors and subtract from elements of the third source (addend) vector without intermediate rounding. Destructively place the negated results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.



FNMLS <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_neg = FALSE;
boolean op3_neg = TRUE;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        bits(esize) element3 = Elem[operand3, e, esize];

        if op1_neg then element1 = FPNeg(element1);
        if op3_neg then element3 = FPNeg(element3);
        Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = Elem[operand3, e, esize];

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

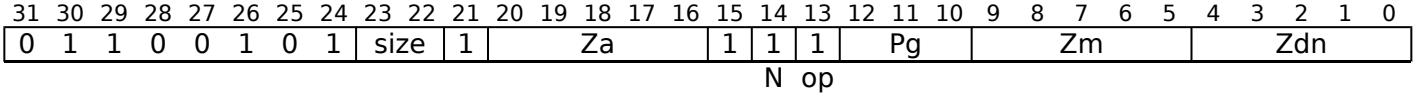
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FNMSB

Floating-point negated fused multiply-subtract vectors (predicated), writing multiplicand [Zdn = -Za + Zdn * Zm]

Multiply the corresponding active floating-point elements of the first and second source vectors and subtract from elements of the third (addend) vector without intermediate rounding. Destructively place the negated results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.



FNMSB <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Za>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
integer a = UInt(Za);
boolean op1_neg = FALSE;
boolean op3_neg = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Za> Is the name of the third source scalable vector register, encoded in the "Za" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = if AnyActiveElement(mask, esize) then Z[a, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        bits(esize) element3 = Elem[operand3, e, esize];

        if op1_neg then element1 = FPNeg(element1);
        if op3_neg then element3 = FPNeg(element3);
        Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

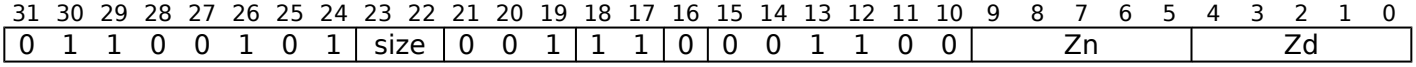
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FRECPE

Floating-point reciprocal estimate (unpredicated)

Find the approximate reciprocal of each floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.



FRECPE <Zd>.<T>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

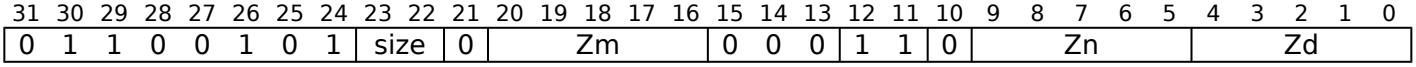
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRecipEstimate(element, FPCR[]);

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FRECPS

Floating-point reciprocal step (unpredicated)

Multiply corresponding floating-point elements of the first and second source vectors, subtract the products from 2.0 without intermediate rounding and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.
This instruction can be used to perform a single Newton-Raphson iteration for calculating the reciprocal of a vector of floating-point values.



FRECPS <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

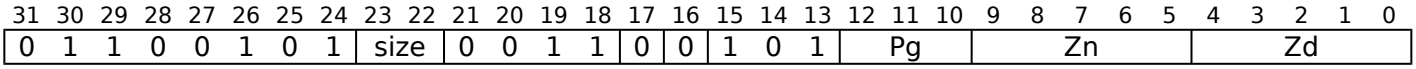
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPRecipStepFused(element1, element2);

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FRECPX

Floating-point reciprocal exponent (predicated)

Invert the exponent and zero the fractional part of each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.
The result of this instruction can be used with FMULX to convert arbitrary elements in mathematical vector space to "unit vectors" or "direction vectors" of length 1.



FRECPX <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = FPRecpX(element, FPCR[]);

Z[d, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FRINT<r>

Floating-point round to integral value (predicated)

Round to an integral floating-point value with the specified rounding option from each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

<r>	Rounding Option
N	to nearest, with ties to even
A	to nearest, with ties away from zero
M	toward minus Infinity
P	toward plus Infinity
Z	toward zero
I	current FPCR rounding mode
X	current FPCR rounding mode, signalling inexact

It has encodings from 7 classes: Current mode , Current mode signalling inexact , Nearest with ties to away , Nearest with ties to even , Toward zero , Toward minus infinity and Toward plus infinity

Current mode

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size		0	0	0	1	1	1	1	0	1	Pg			Zn					Zd				

FRINTI <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

Current mode signalling inexact

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size		0	0	0	1	1	0	1	0	1	Pg			Zn					Zd				

FRINTX <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

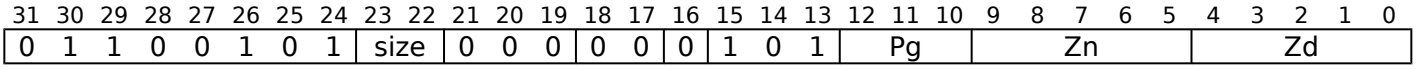
Nearest with ties to away

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size		0	0	0	1	0	0	1	0	1	Pg			Zn					Zd				

FRINTA <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRounding_TIEAWAY;
```

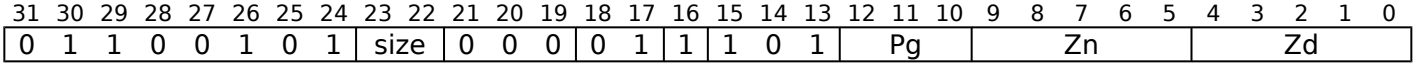
Nearest with ties to even



FRINTN <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRounding_TIEEVEN;
```

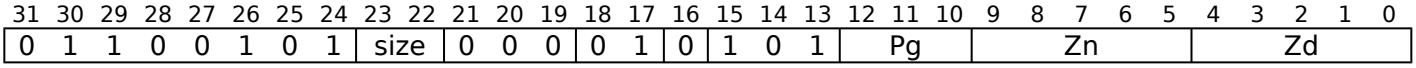
Toward zero



FRINTZ <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRounding_ZERO;
```

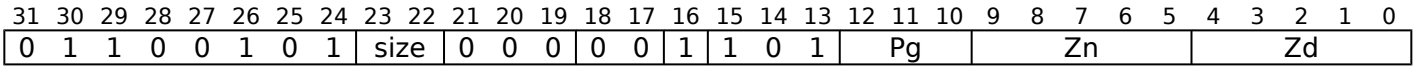
Toward minus infinity



FRINTM <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRounding_NEGINF;
```

Toward plus infinity



FRINTP <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean exact = FALSE;
FPRounding rounding = FPRounding_POSINF;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = FPRoundInt(element, FPCR[], rounding, exact);

Z[d, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

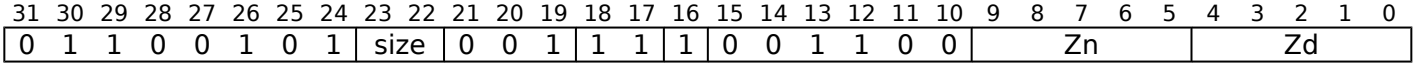
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FRSQRTE

Floating-point reciprocal square root estimate (unpredicated)

Find the approximate reciprocal square root of each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.



FRSQRTE <Zd>.<T>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element = Elem[operand, e, esize];
    Elem[result, e, esize] = FPRSqrtEstimate(element, FPCR[]);

Z[d, VL] = result;
```

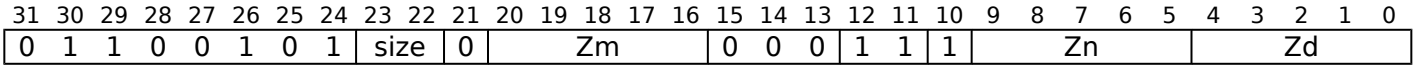
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FRSQRTS

Floating-point reciprocal square root step (unpredicated)

Multiply corresponding floating-point elements of the first and second source vectors, subtract the products from 3.0 and divide the results by 2.0 without any intermediate rounding and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.

This instruction can be used to perform a single Newton-Raphson iteration for calculating the reciprocal square root of a vector of floating-point values.



FRSQRTS <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

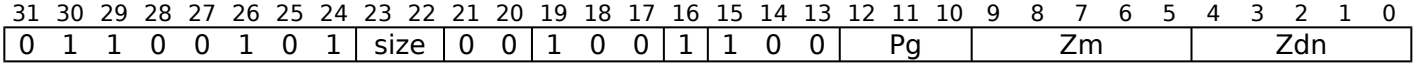
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPRSqrtStepFused(element1, element2);

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FSCALE

Floating-point adjust exponent by vector (predicated)

Multiply the active floating-point elements of the first source vector by 2.0 to the power of the signed integer values in the corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



FSCALE <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        integer element2 = SInt(Elem[operand2, e, esize]);
        Elem[result, e, esize] = FPScale(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

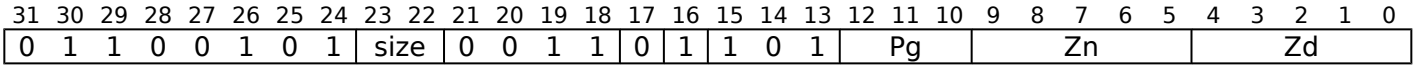
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FSQRT

Floating-point square root (predicated)

Calculate the square root of each active floating-point element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.



FSQRT <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = FPSqrt(element, FPCR[]);

Z[d, VL] = result;
```

Operational information

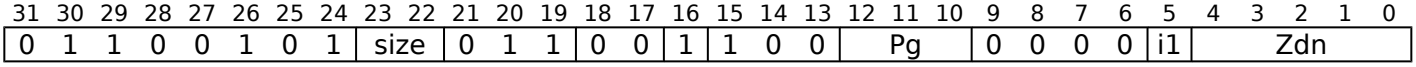
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

FSUB (immediate)

Floating-point subtract immediate (predicated)

Subtract an immediate from each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.5 or +1.0 only. Inactive elements in the destination vector register remain unmodified.



FSUB <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then FPPointFive('0', esize) else FPOne('0', esize);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the floating-point immediate value, encoded in “i1”:

i1	<const>
0	#0.5
1	#1.0

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = FPSub(element1, imm, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

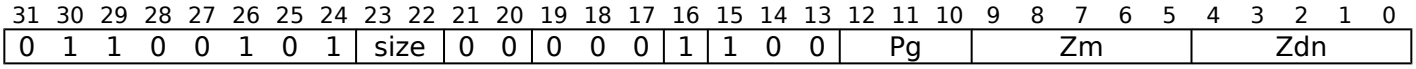
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FSUB (vectors, predicated)

Floating-point subtract vectors (predicated)

Subtract active floating-point elements of the second source vector from corresponding floating-point elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



FSUB <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = FPSub(element1, element2, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

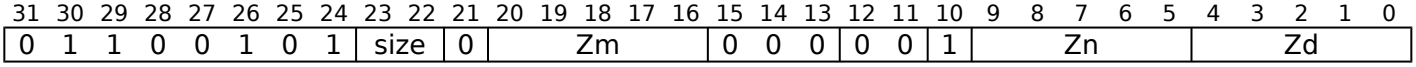
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FSUB (vectors, unpredicated)

Floating-point subtract vectors (unpredicated)

Subtract all floating-point elements of the second source vector from corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.



FSUB <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

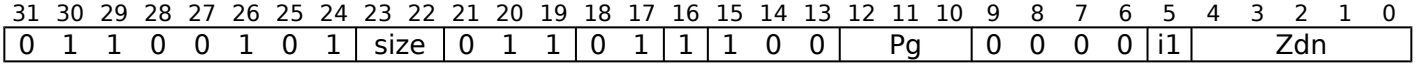
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPSub(element1, element2, FPCR[]);

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FSUBR (immediate)

Floating-point reversed subtract from immediate (predicated)

Reversed subtract from an immediate each active floating-point element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate may take the value +0.5 or +1.0 only. Inactive elements in the destination vector register remain unmodified.



FSUBR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
bits(esize) imm = if i1 == '0' then FPPointFive('0', esize) else FPOne('0', esize);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the floating-point immediate value, encoded in “i1”:

i1	<const>
0	#0.5
1	#1.0

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = FPSub(imm, element1, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

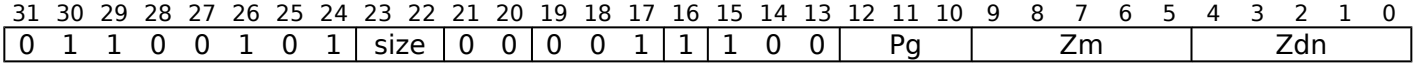
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FSUBR (vectors)

Floating-point reversed subtract vectors (predicated)

Reversed subtract active floating-point elements of the first source vector from corresponding floating-point elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



FSUBR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element2 = Elem[operand2, e, esize];
        Elem[result, e, esize] = FPSub(element2, element1, FPCR[]);
    else
        Elem[result, e, esize] = element1;

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.

- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FTMAD

Floating-point trigonometric multiply-add coefficient

The FTMAD instruction calculates the series terms for either SIN(X) or COS(X), where the argument X has been adjusted to be in the range $-\pi/4 < X \leq \pi/4$.

To calculate the series terms of SIN(X) and COS(X) the initial source operands of FTMAD should be zero in the first source vector and X^2 in the second source vector. The FTMAD instruction is then executed eight times to calculate the sum of eight series terms, which gives a result of sufficient precision.

The FTMAD instruction multiplies each element of the first source vector by the absolute value of the corresponding element of the second source vector and performs a fused addition of each product with a value obtained from a table of hard-wired coefficients, and places the results destructively in the first source vector.

The coefficients are different for SIN(X) and COS(X), and are selected by a combination of the sign bit in the second source element and an immediate index in the range 0 to 7.

Double-precision coefficient table for sin(x) (s2<63> == '0')

Index	Hexadecimal	Decimal	Exact Value
0	3ff0 0000 0000 0000	1.0	= 1/1!
1	bfc5 5555 5555 5543	-0.1666666666666661	> -1/3!
2	3f81 1111 1110 f30c	0.8333333333320002e-02	< 1/5!
3	bf2a 01a0 19b9 2fc6	-0.1984126982840213e-03	> -1/7!
4	3ec7 1de3 51f3 d22b	0.2755731329901505e-05	< 1/9!
5	be5a e5e2 b60f 7b91	-0.2505070584637887e-07	> -1/11!
6	3de5 d840 8868 552f	0.1589413637195215e-09	< 1/13!
7	0000 0000 0000 0000	0.0	> -1/15!

Double-precision coefficient table for cos(x) (s2<63> == '1')

Index	Hexadecimal	Decimal	Exact Value
0	3ff0 0000 0000 0000	1.0	= 1/0!
1	bfe0 0000 0000 0000	-0.5000000000000000	= -1/2!
2	3fa5 5555 5555 5536	0.4166666666666645e-01	< 1/4!
3	bf56 c16c 16c1 3a0b	-0.1388888888886111e-02	> -1/6!
4	3efa 01a0 19b1 e8d8	0.2480158728388683e-04	< 1/8!
5	be92 7e4f 7282 f468	-0.2755731309913950e-06	> -1/10!
6	3e21 ee96 d264 1b13	0.2087558253975872e-08	< 1/12!
7	bda8 f763 80fb b401	-0.1135338700720054e-10	> -1/14!

Single-precision coefficient table for sin(x) (s2<31> == '0')

Index	Hexadecimal	Decimal	Exact Value
0	3f80 0000	1.0	= 1/1!
1	be2a aaab	-1.666666716337e-01	> -1/3!
2	3c08 8886	8.333330973983e-03	< 1/5!
3	b950 08b9	-1.983967522392e-04	> -1/7!
4	3636 9d6d	2.721174723774e-06	< 1/9!
5	0000 0000	0.0	> -1/11!
6	0000 0000	0.0	< 1/13!
7	0000 0000	0.0	> -1/15!

Single-precision coefficient table for cos(x) (s2<31> == '1')

Index	Hexadecimal	Decimal	Exact Value
0	3f80 0000	1.0	= 1/0!
1	bf00 0000	-5.000000000000e-01	= -1/2!
2	3d2a aaa6	4.166664928198e-02	< 1/4!
3	bab6 0705	-1.388759003021e-03	> -1/6!
4	37cd 37cc	2.446388680255e-05	< 1/8!
5	0000 0000	0.0	> -1/10!
6	0000 0000	0.0	< 1/12!
7	0000 0000	0.0	> -1/14!

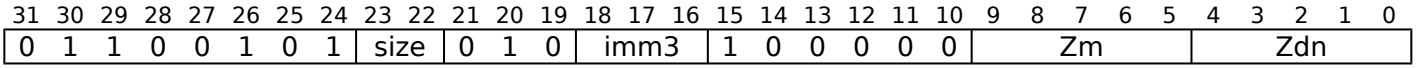
Half-precision coefficient table for sin(x) (s2<15> == '0')

Index	Hexadecimal	Decimal	Exact Value
0	3c00	1.0	= 1/1!
1	b155	-1.666666716337e-01	> -1/3!
2	2030	8.333330973983e-03	< 1/5!
3	0000	0.0	> -1/7!
4	0000	0.0	< 1/9!
5	0000	0.0	> -1/11!
6	0000	0.0	< 1/13!
7	0000	0.0	> -1/15!

Half-precision coefficient table for cos(x) (s2<15> == '1')

Index	Hexadecimal	Decimal	Exact Value
0	3c00	1.0	= 1/0!
1	b800	-5.000000000000e-01	= -1/2!
2	293a	4.166664928198e-02	< 1/4!
3	0000	0.0	> -1/6!
4	0000	0.0	< 1/8!
5	0000	0.0	> -1/10!
6	0000	0.0	< 1/12!
7	0000	0.0	> -1/14!

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.



FTMAD <Zdn>.<T>, <Zdn>.<T>, <Zm>.<T>, #<imm>

```
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
integer imm = UInt(imm3);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<imm> Is the unsigned immediate operand, in the range 0 to 7, encoded in the "imm3" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPTrigMAdd(imm, element1, element2, FPCR[]);

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FTSMUL

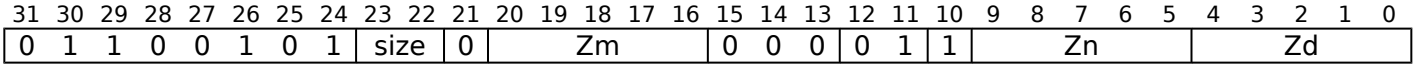
Floating-point trigonometric starting value

The FTSMUL instruction calculates the initial value for the FTMAD instruction. The instruction squares each element in the first source vector and then sets the sign bit to a copy of bit 0 of the corresponding element in the second source register, and places the results in the destination vector. This instruction is unpredicated.

To compute SIN(X) or COS(X) the instruction is executed with elements of the first source vector set to X, adjusted to be in the range $-\pi/4 < X \leq \pi/4$.

The elements of the second source vector hold the corresponding value of the quadrant Q number as an integer not a floating-point value. The value Q satisfies the relationship $(2q-1) \times \pi/4 < X \leq (2q+1) \times \pi/4$.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.



FTSMUL <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPTrigSMul(element1, element2, FPCR[]);

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FTSSEL

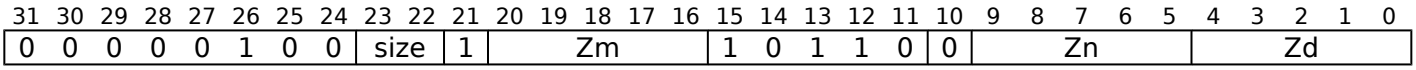
Floating-point trigonometric select coefficient

The FTSSEL instruction selects the coefficient for the final multiplication in the polynomial series approximation. The instruction places the value 1.0 or a copy of the first source vector element in the destination element, depending on bit 0 of the quadrant number Q held in the corresponding element of the second source vector. The sign bit of the destination element is copied from bit 1 of the corresponding value of Q. This instruction is unpredicated.

To compute SIN(X) or COS(X) the instruction is executed with elements of the first source vector set to X, adjusted to be in the range $-\pi/4 < X \leq \pi/4$.

The elements of the second source vector hold the corresponding value of the quadrant Q number as an integer not a floating-point value. The value Q satisfies the relationship $(2q-1) \times \pi/4 < X \leq (2q+1) \times \pi/4$.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.



FTSSEL <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = FPTrigSSel(element1, element2);

Z[d, VL] = result;
```

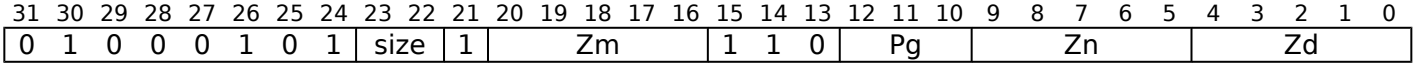
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

HISTCNT

Count matching elements in vector

This instruction compares each active 32 or 64-bit element of the first source vector with all active elements with an element number less than or equal to its own in the second source vector, and places the count of matching elements in the corresponding element of the destination vector. Inactive elements in the destination vector are set to zero.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.



HISTCNT <Zd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE2() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer d = UInt(Zd);
integer n = UInt(Zn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size<0>”:

size<0>	<T>
0	S
1	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

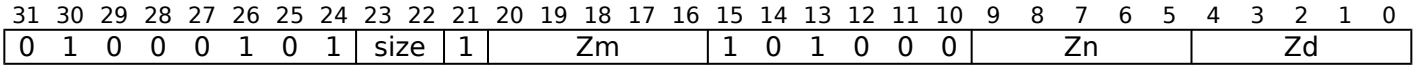
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer count = 0;
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element1 = Elem[operand1, e, esize];
        for i = 0 to e
            if ElemP[mask, i, esize] == '1' then
                bits(esize) element2 = Elem[operand2, i, esize];
                if element1 == element2 then
                    count = count + 1;
    Elem[result, e, esize] = count<esize-1:0>;

Z[d, VL] = result;
```

HISTSEG

Count matching elements in vector segments

This instruction compares each 8-bit byte element of the first source vector with all of the elements in the corresponding 128-bit segment of the second source vector and places the count of matching elements in the corresponding element of the destination vector. This instruction is unpredicated.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.



HISTSEG <Zd>.B, <Zn>.B, <Zm>.B

```
if !HaveSVE2() then UNDEFINED;
if size != '00' then UNDEFINED;
constant integer esize = 8;
integer d = UInt(Zd);
integer n = UInt(Zn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer segments = VL DIV 128;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for b = 0 to segments-1
    for s = 0 to eltspersegment-1
        integer count = 0;
        integer e = eltspersegment * b + s;
        bits(esize) element1 = Elem[operand1, e, esize];
        for i = 0 to eltspersegment-1
            integer e2 = eltspersegment * b + i;
            bits(esize) element2 = Elem[operand2, e2, esize];
            if element1 == element2 then
                count = count + 1;
        Elem[result, e, esize] = count<esize-1:0>;

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

INCB, INCD, INCH, INCW (scalar)

Increment scalar by multiple of predicate constraint element count

Determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination.

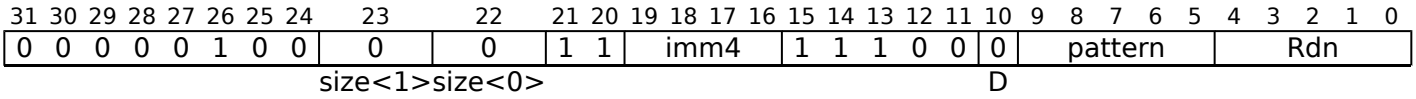
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 4 classes: Byte , Doubleword , Halfword and Word

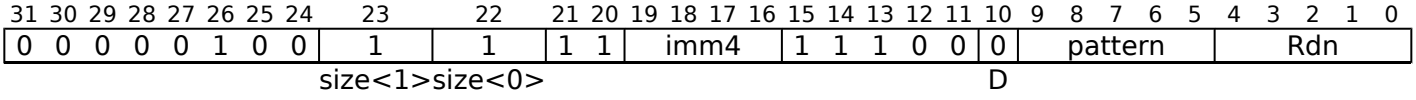
Byte



INCB <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

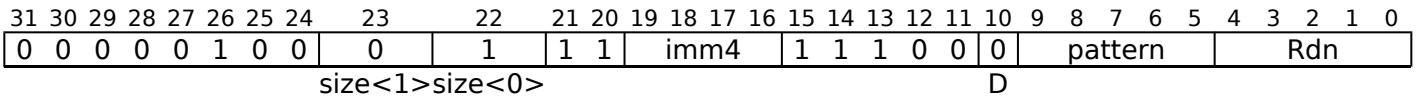
Doubleword



INCD <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

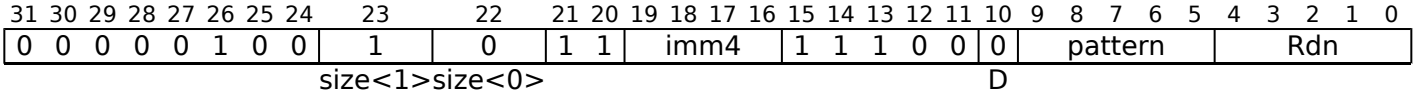
Halfword



INCH <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

Word



INCW <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

Assembler Symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
constant integer VL = CurrentVL;
bits(64) operand1 = X[dn, 64];

X[dn, 64] = operand1 + (count * imm);
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

INCD, INCH, INCW (vector)

Increment vector by multiple of predicate constraint element count

Determines the number of active elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements.

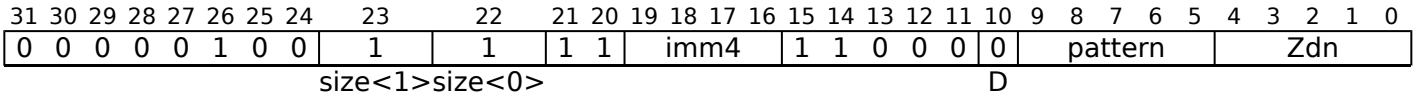
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 3 classes: Doubleword , Halfword and Word

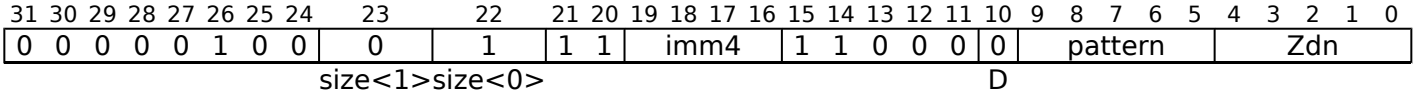
Doubleword



INCD <Zdn>.D{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

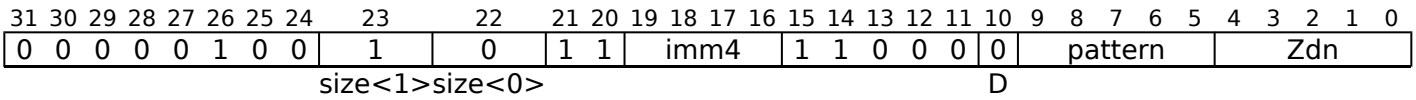
Halfword



INCH <Zdn>.H{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

Word



INCW <Zdn>.S{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    Elem[result, e, esize] = Elem[operand1, e, esize] + (count * imm);

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

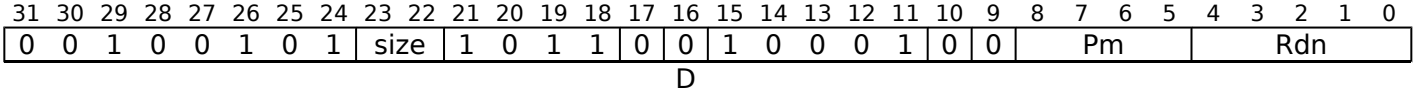
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

INCP (scalar)

Increment scalar by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to increment the scalar destination.



INCP <Xdn>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
```

Assembler Symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) operand1 = X[dn, 64];
bits(PL) operand2 = P[m, PL];
integer count = 0;

for e = 0 to elements-1
    if ElemP[operand2, e, esize] == '1' then
        count = count + 1;

X[dn, 64] = operand1 + count;
```

Operational information

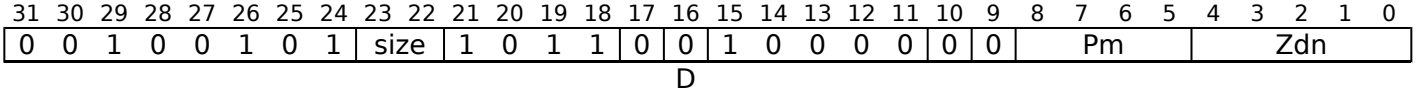
If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

INCP (vector)

Increment vector by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to increment all destination vector elements.
The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.



INCP <Zdn>.<T>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) operand2 = P[m, PL];
bits(VL) result;
integer count = 0;

for e = 0 to elements-1
    if ElemP[operand2, e, esize] == '1' then
        count = count + 1;

for e = 0 to elements-1
    Elem[result, e, esize] = Elem[operand1, e, esize] + count;

Z[dn, VL] = result;
```

Operational information

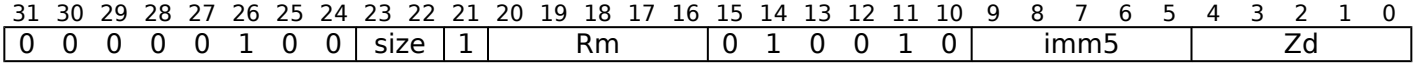
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

INDEX (immediate, scalar)

Create index starting from immediate and incremented by general-purpose register

Populates the destination vector by setting the first element to the first signed immediate integer operand and monotonically incrementing the value by the second signed scalar integer operand for each subsequent element. The scalar source operand is a general-purpose register in which only the least significant bits corresponding to the vector element size are used and any remaining bits are ignored. This instruction is unpredicated.



INDEX <Zd>.<T>, #<imm>, <R><m>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Rm);
integer d = UInt(Zd);
integer imm = SInt(imm5);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<imm> Is the signed immediate operand, in the range -16 to 15, encoded in the "imm5" field.

<R> Is a width specifier, encoded in “size”:

size	<R>
01	W
x0	W
11	X

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(esize) operand2 = X[m, esize];
integer element2 = SInt(operand2);
bits(VL) result;

for e = 0 to elements-1
    integer index = imm + e * element2;
    Elem[result, e, esize] = index<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

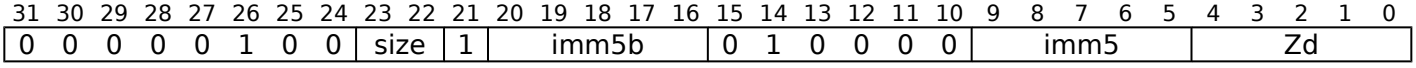
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

INDEX (immediates)

Create index starting from and incremented by immediate

Populates the destination vector by setting the first element to the first signed immediate integer operand and monotonically incrementing the value by the second signed immediate integer operand for each subsequent element. This instruction is unpredicated.



INDEX <Zd>.<T>, #<imm1>, #<imm2>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer d = UInt(Zd);
integer imm1 = SInt(imm5);
integer imm2 = SInt(imm5b);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<imm1> Is the first signed immediate operand, in the range -16 to 15, encoded in the "imm5" field.

<imm2> Is the second signed immediate operand, in the range -16 to 15, encoded in the "imm5b" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) result;

for e = 0 to elements-1
    integer index = imm1 + e * imm2;
    Elem[result, e, esize] = index<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

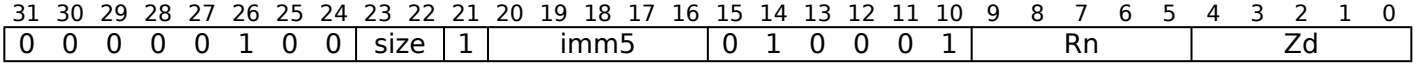
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

INDEX (scalar, immediate)

Create index starting from general-purpose register and incremented by immediate

Populates the destination vector by setting the first element to the first signed scalar integer operand and monotonically incrementing the value by the second signed immediate integer operand for each subsequent element. The scalar source operand is a general-purpose register in which only the least significant bits corresponding to the vector element size are used and any remaining bits are ignored. This instruction is unpredicated.



INDEX <Zd>.<T>, <R><n>, #<imm>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Rn);
integer d = UInt(Zd);
integer imm = SInt(imm5);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<R> Is a width specifier, encoded in “size”:

size	<R>
01	W
x0	W
11	X

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.

<imm> Is the signed immediate operand, in the range -16 to 15, encoded in the "imm5" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(esize) operand1 = X[n, esize];
integer element1 = SInt(operand1);
bits(VL) result;

for e = 0 to elements-1
    integer index = element1 + e * imm;
    Elem[result, e, esize] = index<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

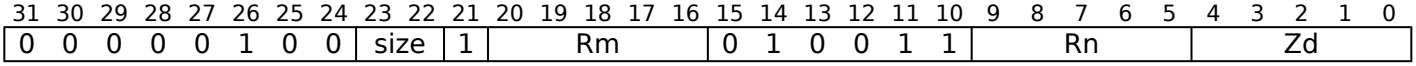
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

INDEX (scalars)

Create index starting from and incremented by general-purpose register

Populates the destination vector by setting the first element to the first signed scalar integer operand and monotonically incrementing the value by the second signed scalar integer operand for each subsequent element. The scalar source operands are general-purpose registers in which only the least significant bits corresponding to the vector element size are used and any remaining bits are ignored. This instruction is unpredicated.



INDEX <Zd>.<T>, <R><n>, <R><m>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<R> Is a width specifier, encoded in “size”:

size	<R>
01	W
x0	W
11	X

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(esize) operand1 = X[n, esize];
integer element1 = SInt(operand1);
bits(esize) operand2 = X[m, esize];
integer element2 = SInt(operand2);
bits(VL) result;

for e = 0 to elements-1
    integer index = element1 + e * element2;
    Elem[result, e, esize] = index<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

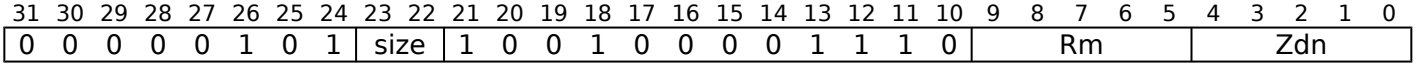
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

INSR (scalar)

Insert general-purpose register in shifted vector

Shift the destination vector left by one element, and then place a copy of the least-significant bits of the general-purpose register in element 0 of the destination vector. This instruction is unpredicated.



INSR <Zdn>.<T>, <R><m>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer m = UInt(Rm);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<R> Is a width specifier, encoded in “size”:

size	<R>
01	W
x0	W
11	X

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(VL) dest = Z[dn, VL];
bits(esize) src = X[m, esize];
Z[dn, VL] = dest<(VL-esize)-1:0> : src;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

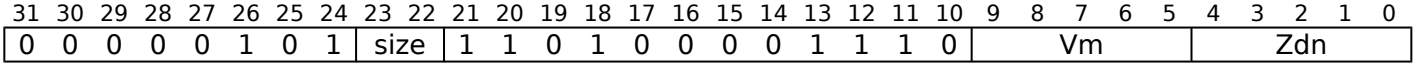
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

INSR (SIMD&FP scalar)

Insert SIMD&FP scalar register in shifted vector

Shift the destination vector left by one element, and then place a copy of the SIMD&FP scalar register in element 0 of the destination vector. This instruction is unpredicated.



INSR <Zdn>.<T>, <V><m>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer m = UInt(Vm);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<m> Is the number [0-31] of the source SIMD&FP register, encoded in the "Vm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(VL) dest = Z[dn, VL];
bits(esize) src = V[m, esize];
Z[dn, VL] = dest<(VL-esize)-1:0> : src;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

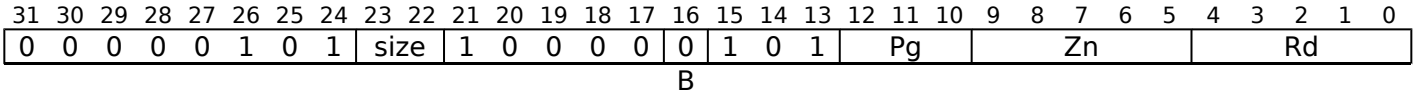
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LASTA (scalar)

Extract element after last to general-purpose register

If there is an active element then extract the element after the last active element modulo the number of elements from the final source vector register. If there are no active elements, extract element zero. Then zero-extend and place the extracted element in the destination general-purpose register.



LASTA <R><d>, <Pg>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = if esize < 64 then 32 else 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Rd);
boolean isBefore = FALSE;
```

Assembler Symbols

<R> Is a width specifier, encoded in “size”:

size	<R>
01	W
x0	W
11	X

<d> Is the number [0-30] of the destination general-purpose register or the name ZR (31), encoded in the "Rd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = Z[n, VL];
bits(rsize) result;
integer last = LastActiveElement(mask, esize);

if isBefore then
    if last < 0 then last = elements - 1;
else
    last = last + 1;
    if last >= elements then last = 0;
result = ZeroExtend(Elem[operand, last, esize], rsize);

X[d, rsize] = result;
```

Operational information

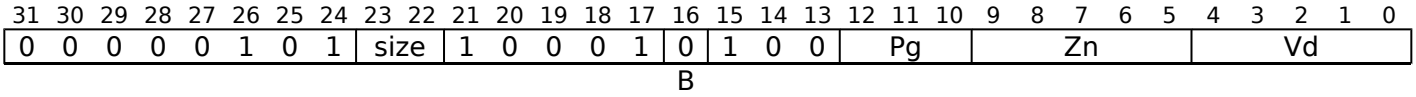
If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LASTA (SIMD&FP scalar)

Extract element after last to SIMD&FP scalar register

If there is an active element then extract the element after the last active element modulo the number of elements from the final source vector register. If there are no active elements, extract element zero. Then place the extracted element in the destination SIMD&FP scalar register.



LASTA <V><d>, <Pg>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean isBefore = FALSE;
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

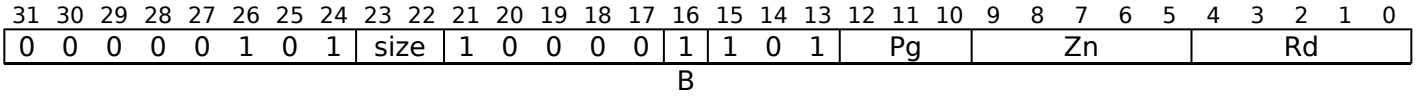
Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = Z[n, VL];
integer last = LastActiveElement(mask, esize);

if isBefore then
    if last < 0 then last = elements - 1;
else
    last = last + 1;
    if last >= elements then last = 0;
V[d, esize] = Elem[operand, last, esize];
```

LASTB (scalar)

Extract last element to general-purpose register

If there is an active element then extract the last active element from the final source vector register. If there are no active elements, extract the highest-numbered element. Then zero-extend and place the extracted element in the destination general-purpose register.



LASTB <R><d>, <Pg>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = if esize < 64 then 32 else 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Rd);
boolean isBefore = TRUE;
```

Assembler Symbols

<R> Is a width specifier, encoded in “size”:

size	<R>
01	W
x0	W
11	X

<d> Is the number [0-30] of the destination general-purpose register or the name ZR (31), encoded in the "Rd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = Z[n, VL];
bits(rsize) result;
integer last = LastActiveElement(mask, esize);

if isBefore then
    if last < 0 then last = elements - 1;
else
    last = last + 1;
    if last >= elements then last = 0;
result = ZeroExtend(Elem[operand, last, esize], rsize);

X[d, rsize] = result;
```

Operational information

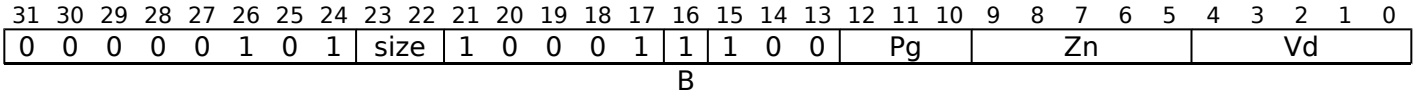
If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LASTB (SIMD&FP scalar)

Extract last element to SIMD&FP scalar register

If there is an active element then extract the last active element from the final source vector register. If there are no active elements, extract the highest-numbered element. Then place the extracted element in the destination SIMD&FP register.



LASTB <V><d>, <Pg>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean isBefore = TRUE;
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

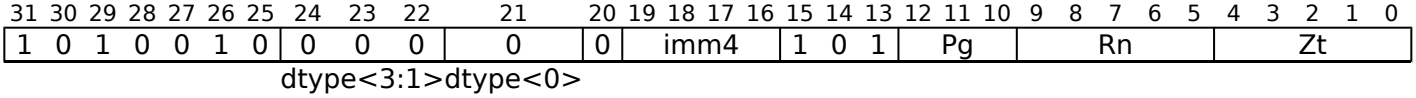
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = Z[n, VL];
integer last = LastActiveElement(mask, esize);

if isBefore then
    if last < 0 then last = elements - 1;
else
    last = last + 1;
    if last >= elements then last = 0;
V[d, esize] = Elem[operand, last, esize];
```

LD1B (scalar plus immediate)

Contiguous load unsigned bytes to vector (immediate index)

Contiguous load of unsigned bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.
It has encodings from 4 classes: 8-bit element , 16-bit element , 32-bit element and 64-bit element

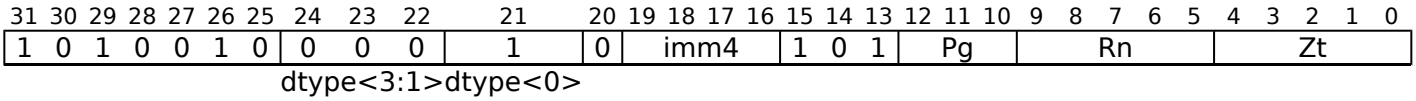
8-bit element



LD1B { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

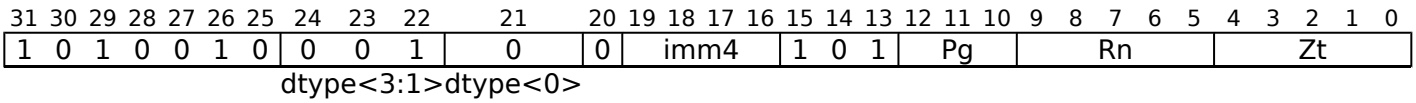
16-bit element



LD1B { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

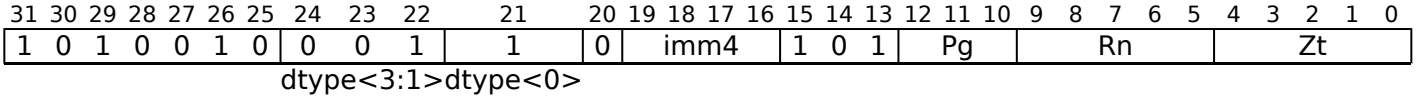
32-bit element



LD1B { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

64-bit element



LD1B { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

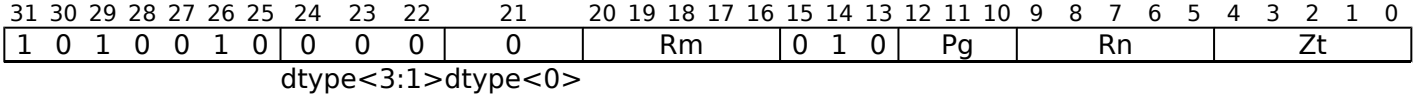
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1B (scalar plus scalar)

Contiguous load unsigned bytes to vector (scalar index)

Contiguous load of unsigned bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.
It has encodings from 4 classes: 8-bit element , 16-bit element , 32-bit element and 64-bit element

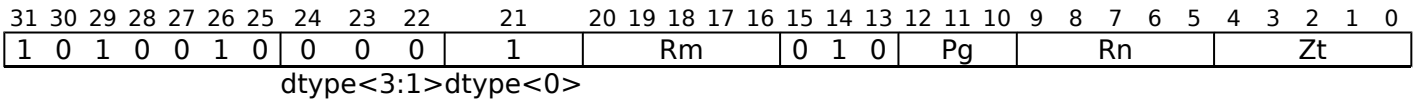
8-bit element



LD1B { <Zt>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
constant integer msize = 8;
boolean unsigned = TRUE;
```

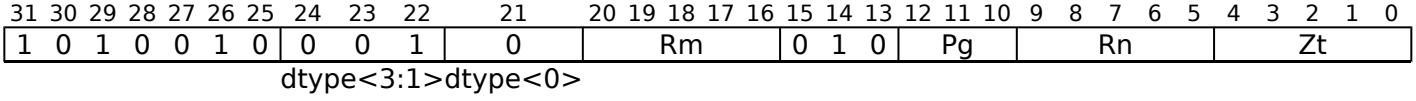
16-bit element



LD1B { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 8;
boolean unsigned = TRUE;
```

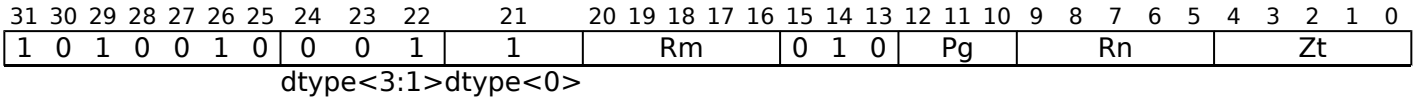
32-bit element



LD1B { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = TRUE;
```

64-bit element



LD1B { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

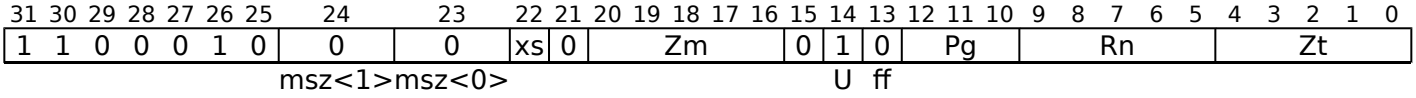
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1B (scalar plus vector)

Gather load unsigned bytes to vector (vector index)

Gather load of unsigned bytes to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 3 classes: 32-bit unpacked unscaled offset , 32-bit unscaled offset and 64-bit unscaled offset

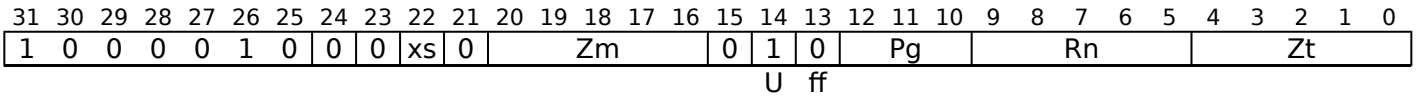
32-bit unpacked unscaled offset



LD1B { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

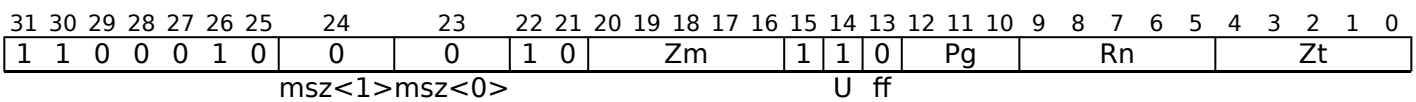
32-bit unscaled offset



LD1B { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

64-bit unscaled offset



LD1B { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

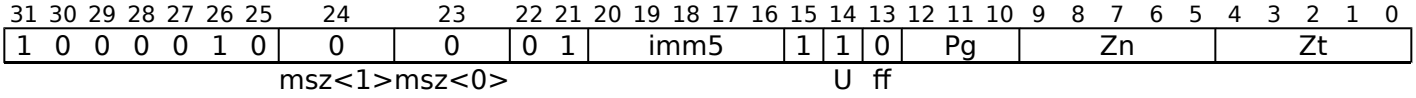
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

LD1B (vector plus immediate)

Gather load unsigned bytes to vector (immediate index)

Gather load of unsigned bytes to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 2 classes: 32-bit element and 64-bit element

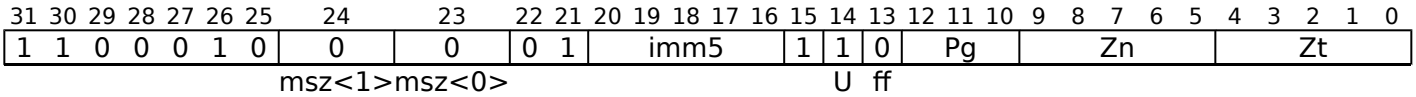
32-bit element



LD1B { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

64-bit element



LD1B { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm>	Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.

Operation

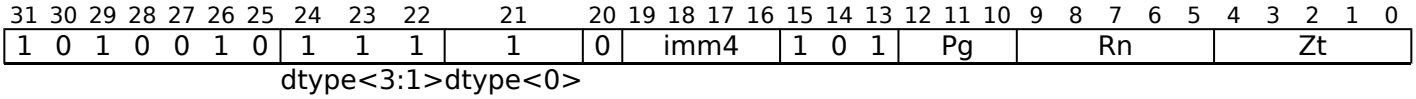
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1D (scalar plus immediate)

Contiguous load doublewords to vector (immediate index)

Contiguous load of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.



LD1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

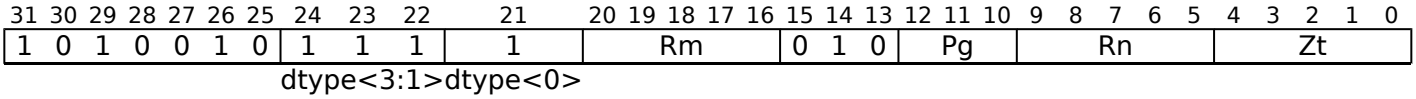
Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

LD1D (scalar plus scalar)

Contiguous load doublewords to vector (scalar index)

Contiguous load of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.



LD1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1D (scalar plus vector)

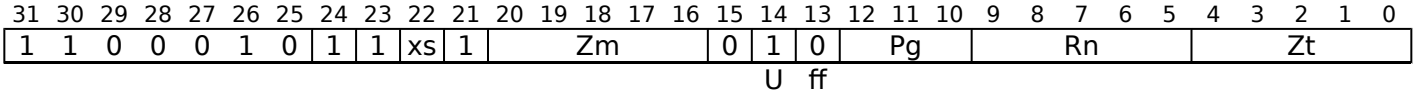
Gather load doublewords to vector (vector index)

Gather load of doublewords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 8. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

It has encodings from 4 classes: 32-bit unpacked scaled offset , 32-bit unpacked unscaled offset , 64-bit scaled offset and 64-bit unscaled offset

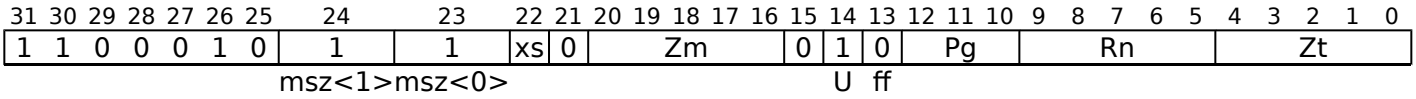
32-bit unpacked scaled offset



LD1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #3]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 3;
```

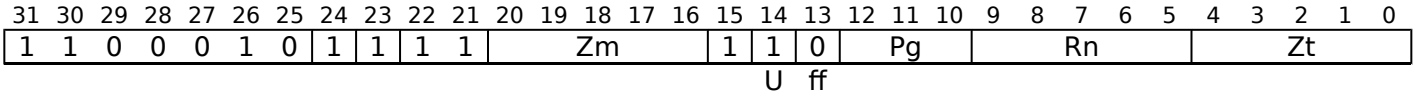
32-bit unpacked unscaled offset



LD1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

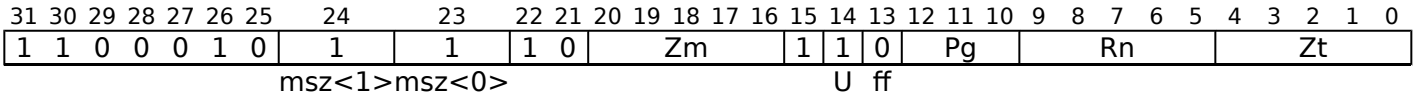
64-bit scaled offset



LD1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #3]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 3;
```

64-bit unscaled offset



LD1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

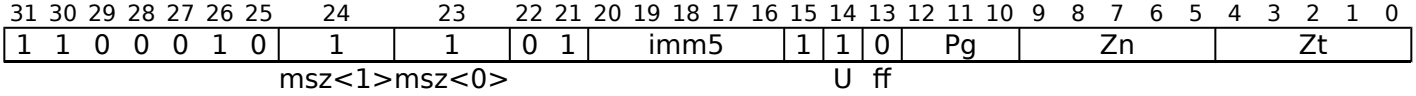
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1D (vector plus immediate)

Gather load doublewords to vector (immediate index)

Gather load of doublewords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 8 in the range 0 to 248. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.



LD1D { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm>	Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 248, defaulting to 0, encoded in the "imm5" field.

Operation

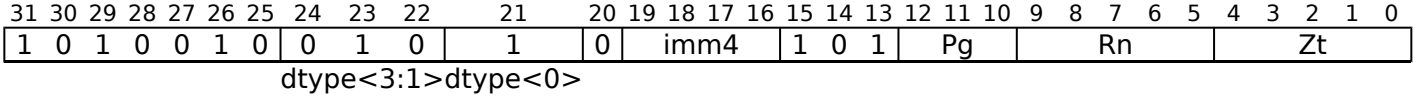
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

LD1H (scalar plus immediate)

Contiguous load unsigned halfwords to vector (immediate index)

Contiguous load of unsigned halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.
It has encodings from 3 classes: 16-bit element , 32-bit element and 64-bit element

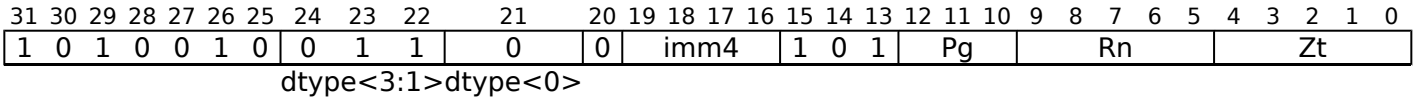
16-bit element



LD1H { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

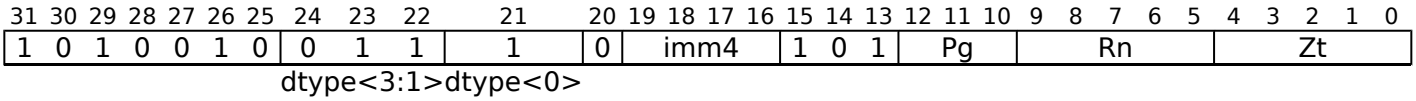
32-bit element



LD1H { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

64-bit element



LD1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

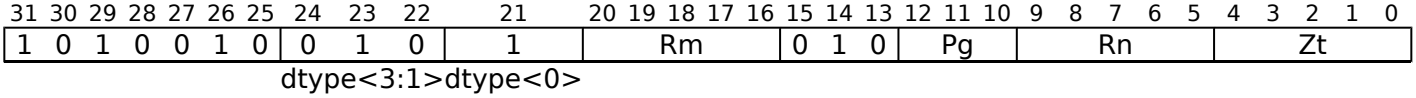
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1H (scalar plus scalar)

Contiguous load unsigned halfwords to vector (scalar index)

Contiguous load of unsigned halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.
It has encodings from 3 classes: 16-bit element , 32-bit element and 64-bit element

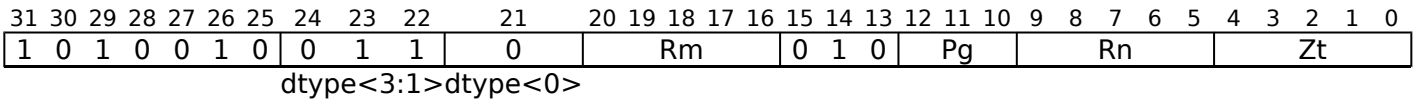
16-bit element



LD1H { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 16;
boolean unsigned = TRUE;
```

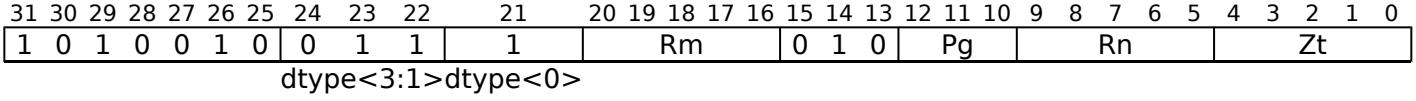
32-bit element



LD1H { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = TRUE;
```

64-bit element



LD1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1H (scalar plus vector)

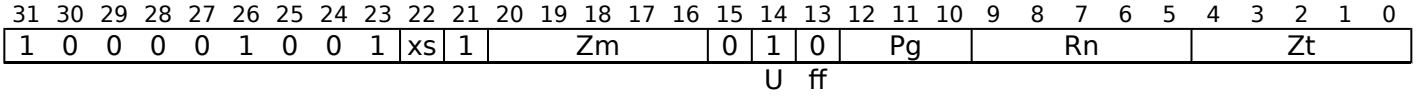
Gather load unsigned halfwords to vector (vector index)

Gather load of unsigned halfwords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

It has encodings from 6 classes: 32-bit scaled offset , 32-bit unpacked scaled offset , 32-bit unpacked unscaled offset , 32-bit unscaled offset , 64-bit scaled offset and 64-bit unscaled offset

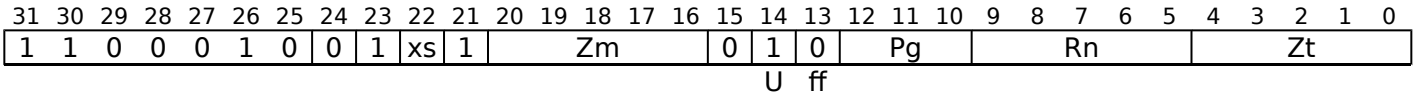
32-bit scaled offset



LD1H { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #1]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```

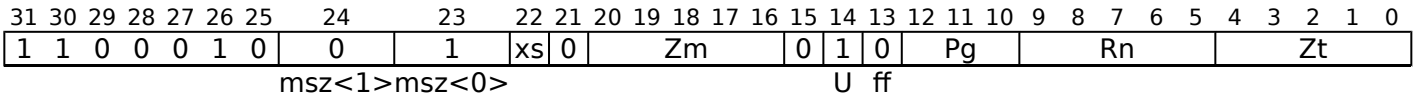
32-bit unpacked scaled offset



LD1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #1]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```

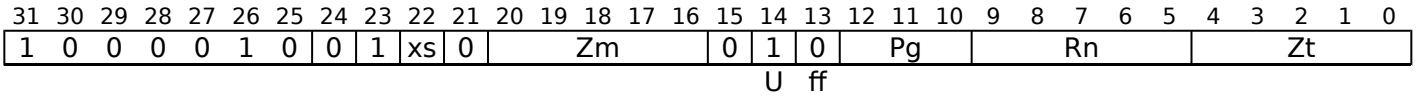
32-bit unpacked unscaled offset



LD1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

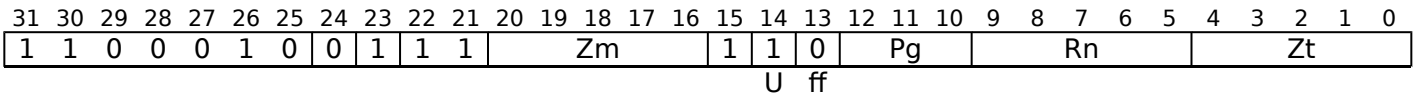
32-bit unscaled offset



LD1H { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

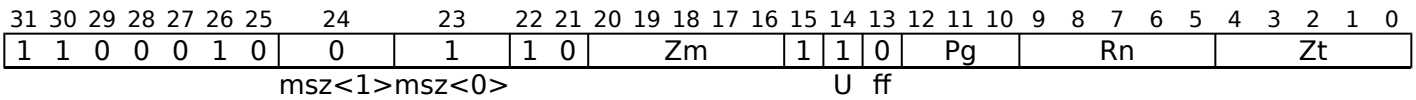
64-bit scaled offset



LD1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #1]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 1;
```

64-bit unscaled offset



LD1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

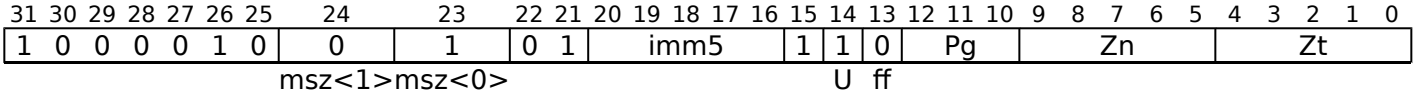
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

LD1H (vector plus immediate)

Gather load unsigned halfwords to vector (immediate index)

Gather load of unsigned halfwords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 2 classes: 32-bit element and 64-bit element

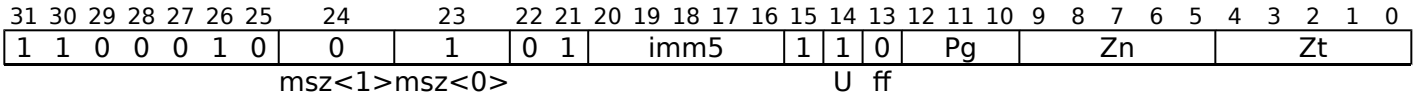
32-bit element



LD1H { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

64-bit element



LD1H { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm>	Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the "imm5" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

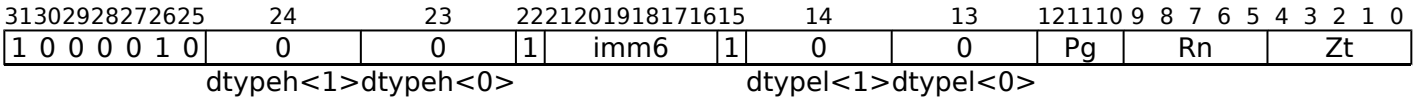
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1RB

Load and broadcast unsigned byte to vector

Load a single unsigned byte from a memory address generated by a 64-bit scalar base address plus an immediate offset which is in the range 0 to 63.
Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.
It has encodings from 4 classes: 8-bit element , 16-bit element , 32-bit element and 64-bit element

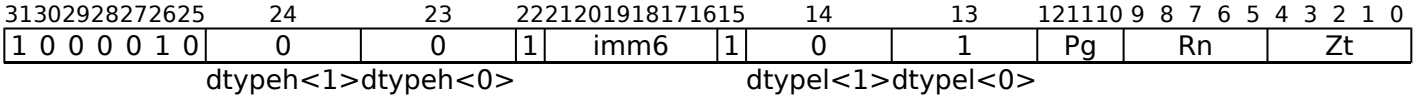
8-bit element



LD1RB { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

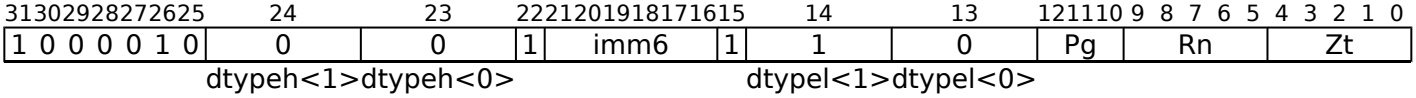
16-bit element



LD1RB { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

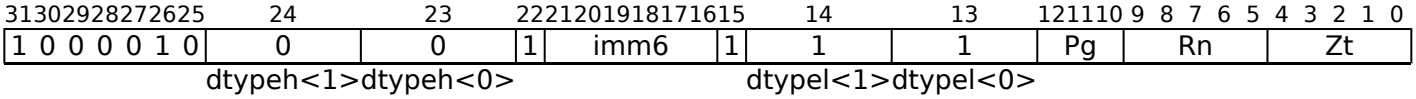
32-bit element



LD1RB { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

64-bit element



LD1RB { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional unsigned immediate byte offset, in the range 0 to 63, defaulting to 0, encoded in the "imm6" field.

Operation

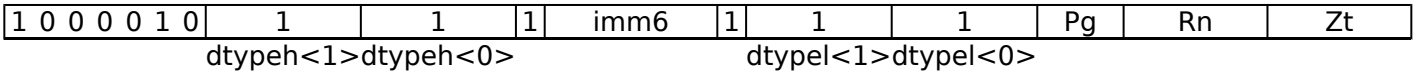
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    bits(64) addr = base + offset * mbytes;
    data = Mem[addr, mbytes, AccType_SVE];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1RD

Load and broadcast doubleword to vector

Load a single doubleword from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 8 in the range 0 to 504.
Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.



LD1RD { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 504, defaulting to 0, encoded in the "imm6" field.

Operation

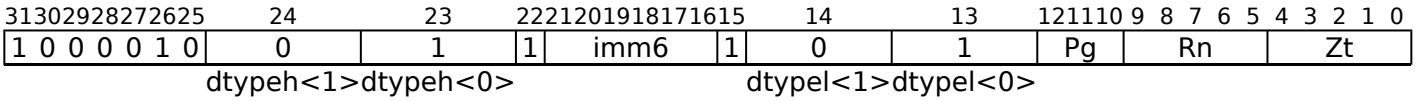
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    bits(64) addr = base + offset * mbytes;
    data = Mem[addr, mbytes, AccType_SVE];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

LD1RH

Load and broadcast unsigned halfword to vector

Load a single unsigned halfword from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 2 in the range 0 to 126.
Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.
It has encodings from 3 classes: 16-bit element , 32-bit element and 64-bit element

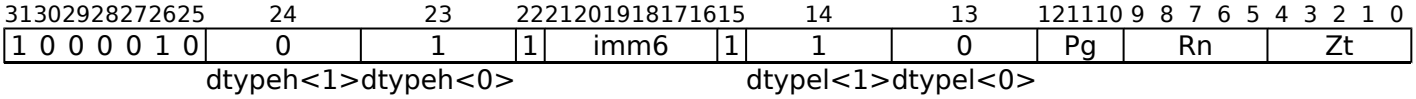
16-bit element



LD1RH { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

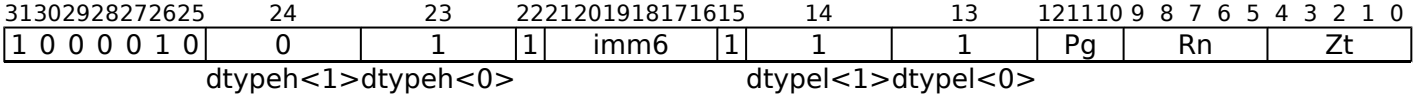
32-bit element



LD1RH { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

64-bit element



LD1RH { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 126, defaulting to 0, encoded in the "imm6" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    bits(64) addr = base + offset * mbytes;
    data = Mem[addr, mbytes, AccType_SVE];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

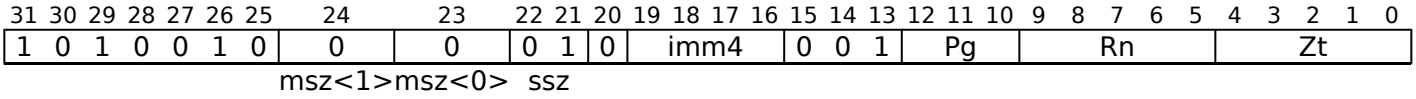
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1ROB (scalar plus immediate)

Contiguous load and replicate thirty-two bytes (immediate index)

Load thirty-two contiguous bytes to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 32 in the range -256 to +224 added to the base address.
Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.
The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.
Only the first thirty-two predicate elements are used and higher numbered predicate elements are ignored.
ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE
(FEAT_F64MM)



LD1ROB { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate byte offset, a multiple of 32 in the range -256 to 224, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
if VL < 256 then UNDEFINED;
constant integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low bits only
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = ZeroExtend(Replicate(result, VL DIV 256), VL);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1ROB (scalar plus scalar)

Contiguous load and replicate thirty-two bytes (scalar index)

Load thirty-two contiguous bytes to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

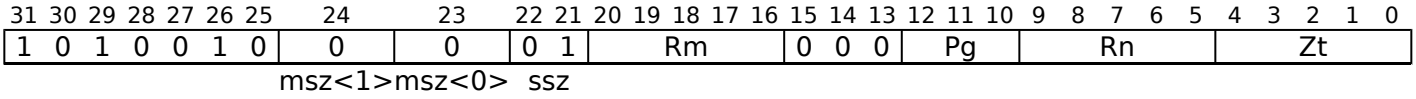
Only the first thirty-two predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE

(FEAT_F64MM)



LD1ROB { <Zt>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]

```
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
if VL < 256 then UNDEFINED;
constant integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low bits only
bits(64) offset;
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = ZeroExtend(Replicate(result, VL DIV 256), VL);
```

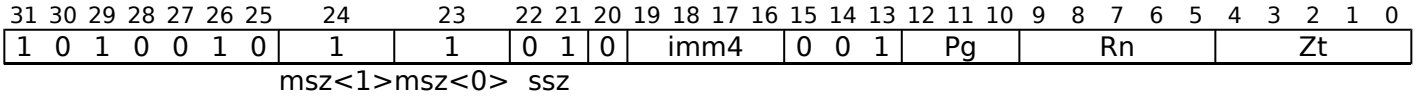
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1ROD (scalar plus immediate)

Contiguous load and replicate four doublewords (immediate index)

Load four contiguous doublewords to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 32 in the range -256 to +224 added to the base address.
Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.
The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.
Only the first four predicate elements are used and higher numbered predicate elements are ignored.
ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE
(FEAT_F64MM)



LD1ROD { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate byte offset, a multiple of 32 in the range -256 to 224, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
if VL < 256 then UNDEFINED;
constant integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low bits only
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = ZeroExtend(Replicate(result, VL DIV 256), VL);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1ROD (scalar plus scalar)

Contiguous load and replicate four doublewords (scalar index)

Load four contiguous doublewords to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 8 and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

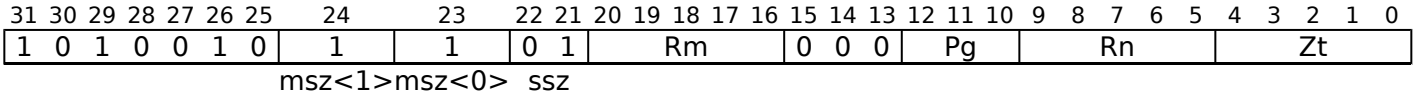
Only the first four predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE

(FEAT_F64MM)



LD1ROD { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
if VL < 256 then UNDEFINED;
constant integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low bits only
bits(64) offset;
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = ZeroExtend(Replicate(result, VL DIV 256), VL);
```

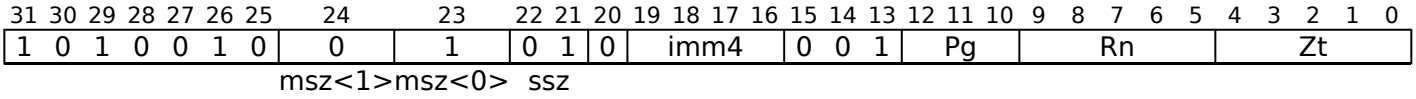
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1ROH (scalar plus immediate)

Contiguous load and replicate sixteen halfwords (immediate index)

Load sixteen contiguous halfwords to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 32 in the range -256 to +224 added to the base address.
Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.
The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.
Only the first sixteen predicate elements are used and higher numbered predicate elements are ignored.
ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE
(FEAT_F64MM)



LD1ROH { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate byte offset, a multiple of 32 in the range -256 to 224, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
if VL < 256 then UNDEFINED;
constant integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low bits only
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = ZeroExtend(Replicate(result, VL DIV 256), VL);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1ROH (scalar plus scalar)

Contiguous load and replicate sixteen halfwords (scalar index)

Load sixteen contiguous halfwords to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 2 and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

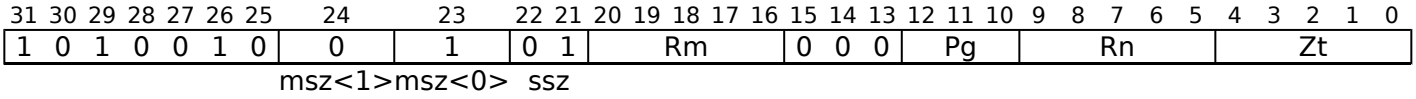
Only the first sixteen predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE

(FEAT_F64MM)



LD1ROH { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
if VL < 256 then UNDEFINED;
constant integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low bits only
bits(64) offset;
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = ZeroExtend(Replicate(result, VL DIV 256), VL);
```

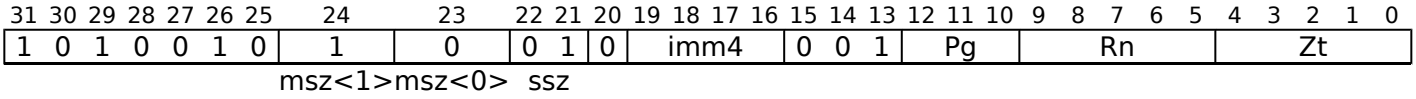
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1ROW (scalar plus immediate)

Contiguous load and replicate eight words (immediate index)

Load eight contiguous words to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 32 in the range -256 to +224 added to the base address.
Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.
The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.
Only the first eight predicate elements are used and higher numbered predicate elements are ignored.
ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE
(FEAT_F64MM)



LD1ROW { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate byte offset, a multiple of 32 in the range -256 to 224, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
if VL < 256 then UNDEFINED;
constant integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low bits only
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = ZeroExtend(Replicate(result, VL DIV 256), VL);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1ROW (scalar plus scalar)

Contiguous load and replicate eight words (scalar index)

Load eight contiguous words to elements of a 256-bit (octaword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 4 and added to the base address.

Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero.

The resulting 256-bit vector is then replicated to fill the destination vector. The instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits in the destination vector are set to zero.

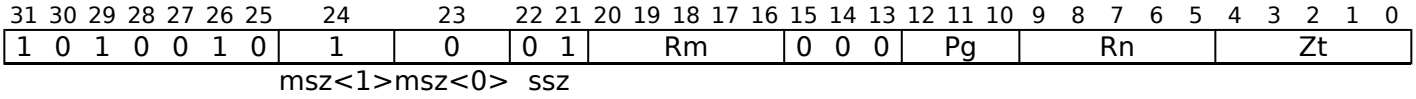
Only the first eight predicate elements are used and higher numbered predicate elements are ignored.

ID_AA64ZFR0_EL1.F64MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE

(FEAT_F64MM)



LD1ROW { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

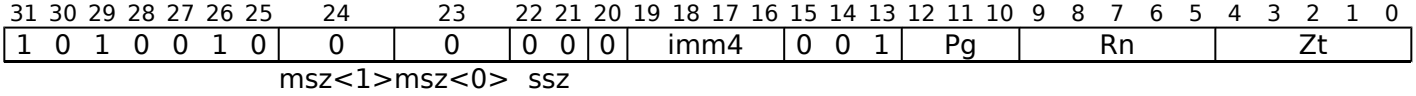
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
if VL < 256 then UNDEFINED;
constant integer elements = 256 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low bits only
bits(64) offset;
bits(256) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = ZeroExtend(Replicate(result, VL DIV 256), VL);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1RQB (scalar plus immediate)

Contiguous load and replicate sixteen bytes (immediate index)

Load sixteen contiguous bytes to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 16 in the range -128 to +112 added to the base address.
Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first sixteen predicate elements are used and higher numbered predicate elements are ignored.



LD1RQB { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate byte offset, a multiple of 16 in the range -128 to 112, defaulting to 0, encoded in the "imm4" field.

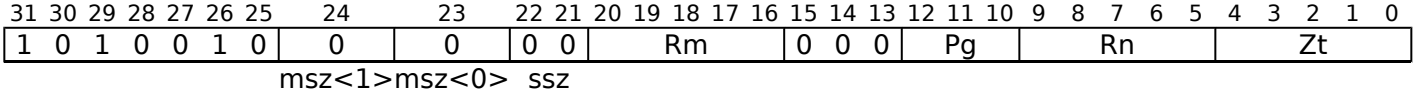
Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = 128 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low 16 bits only
bits(128) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (offset * 16) + (e * mbytes);
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = Replicate(result, VL DIV 128);
```

LD1RQB (scalar plus scalar)

Contiguous load and replicate sixteen bytes (scalar index)

Load sixteen contiguous bytes to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is added to the base address.
Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first sixteen predicate elements are used and higher numbered predicate elements are ignored.



LD1RQB { <Zt>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

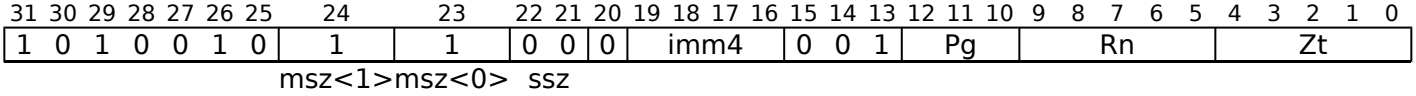
Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = 128 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low 16 bits only
bits(64) offset;
bits(128) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = Replicate(result, VL DIV 128);
```

LD1RQD (scalar plus immediate)

Contiguous load and replicate two doublewords (immediate index)

Load two contiguous doublewords to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 16 in the range -128 to +112 added to the base address.
Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first two predicate elements are used and higher numbered predicate elements are ignored.



LD1RQD { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate byte offset, a multiple of 16 in the range -128 to 112, defaulting to 0, encoded in the "imm4" field.

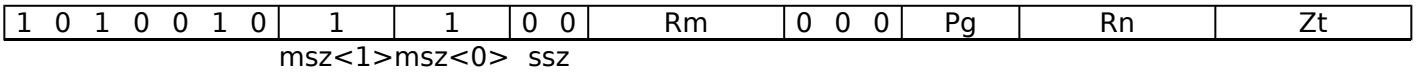
Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = 128 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low 16 bits only
bits(128) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (offset * 16) + (e * mbytes);
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = Replicate(result, VL DIV 128);
```

LD1RQD (scalar plus scalar)

Contiguous load and replicate two doublewords (scalar index)

Load two contiguous doublewords to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 8 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first two predicate elements are used and higher numbered predicate elements are ignored.



LD1RQD { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

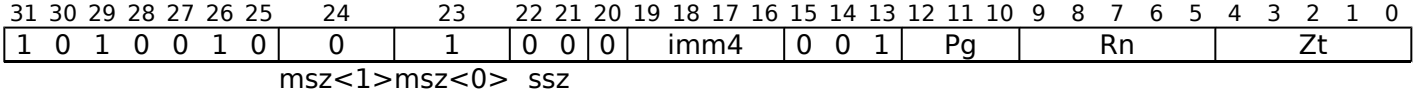
Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = 128 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low 16 bits only
bits(64) offset;
bits(128) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = Replicate(result, VL DIV 128);
```

LD1RQH (scalar plus immediate)

Contiguous load and replicate eight halfwords (immediate index)

Load eight contiguous halfwords to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 16 in the range -128 to +112 added to the base address.
Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first eight predicate elements are used and higher numbered predicate elements are ignored.



LD1RQH { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate byte offset, a multiple of 16 in the range -128 to 112, defaulting to 0, encoded in the "imm4" field.

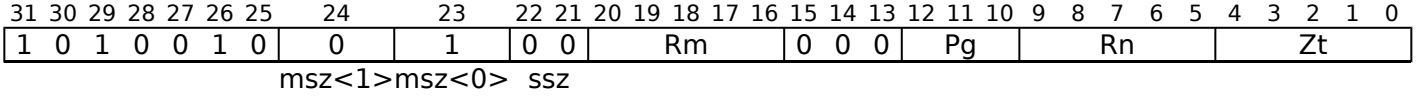
Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = 128 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low 16 bits only
bits(128) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (offset * 16) + (e * mbytes);
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = Replicate(result, VL DIV 128);
```

LD1RQH (scalar plus scalar)

Contiguous load and replicate eight halfwords (scalar index)

Load eight contiguous halfwords to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 2 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first eight predicate elements are used and higher numbered predicate elements are ignored.



LD1RQH { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

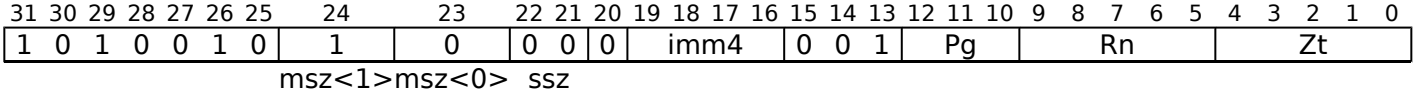
Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = 128 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low 16 bits only
bits(64) offset;
bits(128) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = Replicate(result, VL DIV 128);
```

LD1RQW (scalar plus immediate)

Contiguous load and replicate four words (immediate index)

Load four contiguous words to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and immediate index that is a multiple of 16 in the range -128 to +112 added to the base address.
Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first four predicate elements are used and higher numbered predicate elements are ignored.



LD1RQW { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate byte offset, a multiple of 16 in the range -128 to 112, defaulting to 0, encoded in the "imm4" field.

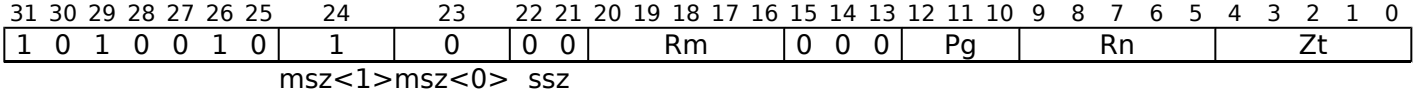
Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = 128 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low 16 bits only
bits(128) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (offset * 16) + (e * mbytes);
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = Replicate(result, VL DIV 128);
```

LD1RQW (scalar plus scalar)

Contiguous load and replicate four words (scalar index)

Load four contiguous words to elements of a short, 128-bit (quadword) vector from the memory address generated by a 64-bit scalar base address and scalar index which is multiplied by 4 and added to the base address.
Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero. The resulting short vector is then replicated to fill the long destination vector. Only the first four predicate elements are used and higher numbered predicate elements are ignored.



LD1RQW { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

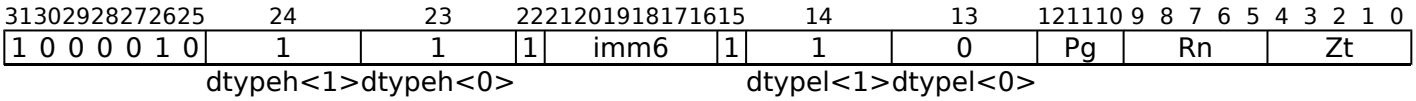
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = 128 DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL]; // low 16 bits only
bits(64) offset;
bits(128) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVE];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = Replicate(result, VL DIV 128);
```

LD1RSB

Load and broadcast signed byte to vector

Load a single signed byte from a memory address generated by a 64-bit scalar base address plus an immediate offset which is in the range 0 to 63.
Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.
It has encodings from 3 classes: 16-bit element , 32-bit element and 64-bit element

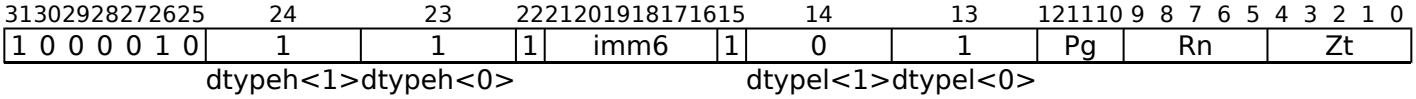
16-bit element



LD1RSB { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm6);
```

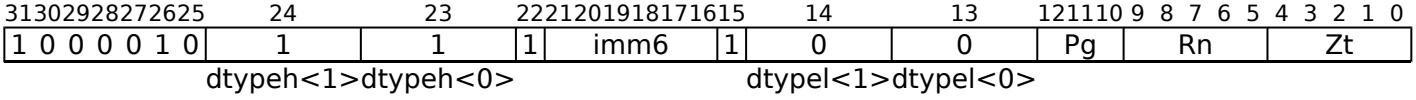
32-bit element



LD1RSB { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm6);
```

64-bit element



LD1RSB { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm6);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional unsigned immediate byte offset, in the range 0 to 63, defaulting to 0, encoded in the "imm6" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    bits(64) addr = base + offset * mbytes;
    data = Mem[addr, mbytes, AccType_SVE];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

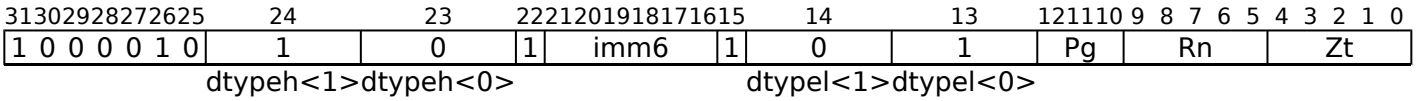
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1RSH

Load and broadcast signed halfword to vector

Load a single signed halfword from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 2 in the range 0 to 126.
Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.
It has encodings from 2 classes: 32-bit element and 64-bit element

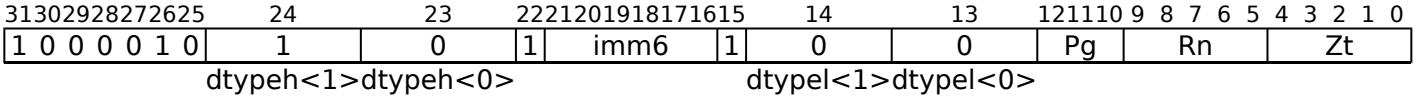
32-bit element



LD1RSH { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm6);
```

64-bit element



LD1RSH { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm6);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 126, defaulting to 0, encoded in the "imm6" field.

Operation

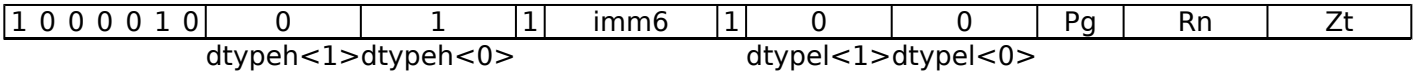
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    bits(64) addr = base + offset * mbytes;
    data = Mem[addr, mbytes, AccType_SVE];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1RSW

Load and broadcast signed word to vector

Load a single signed word from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 4 in the range 0 to 252.
Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.



LD1RSW { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = FALSE;
integer offset = UInt(imm6);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 252, defaulting to 0, encoded in the "imm6" field.

Operation

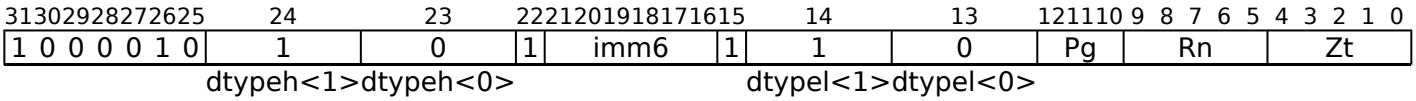
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    bits(64) addr = base + offset * mbytes;
    data = Mem[addr, mbytes, AccType_SVE];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

LD1RW

Load and broadcast unsigned word to vector

Load a single unsigned word from a memory address generated by a 64-bit scalar base address plus an immediate offset which is a multiple of 4 in the range 0 to 252.
Broadcast the loaded data into all active elements of the destination vector, setting the inactive elements to zero. If all elements are inactive then the instruction will not perform a read from Device memory or cause a data abort.
It has encodings from 2 classes: 32-bit element and 64-bit element

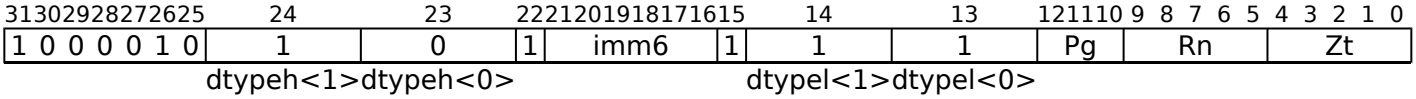
32-bit element



LD1RW { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

64-bit element



LD1RW { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm6);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 252, defaulting to 0, encoded in the "imm6" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    bits(64) addr = base + offset * mbytes;
    data = Mem[addr, mbytes, AccType_SVE];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

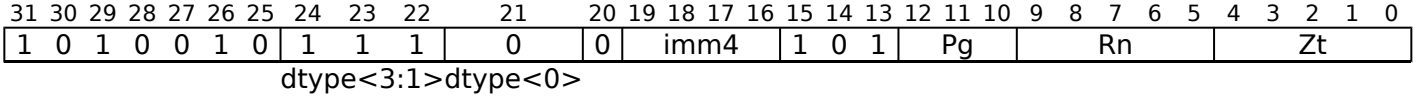
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1SB (scalar plus immediate)

Contiguous load signed bytes to vector (immediate index)

Contiguous load of signed bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.
It has encodings from 3 classes: 16-bit element , 32-bit element and 64-bit element

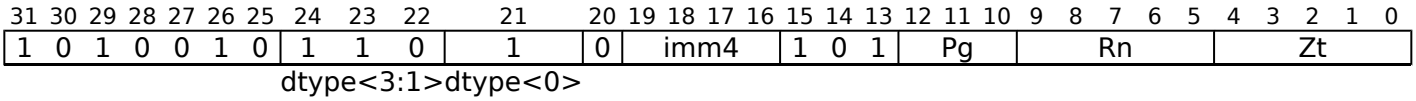
16-bit element



LD1SB { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
```

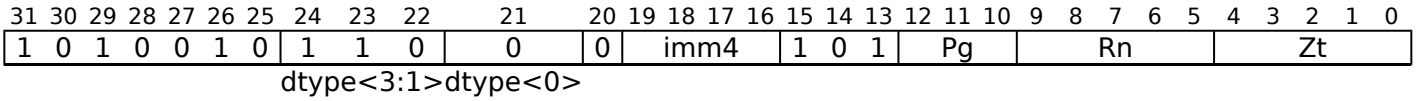
32-bit element



LD1SB { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
```

64-bit element



LD1SB { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

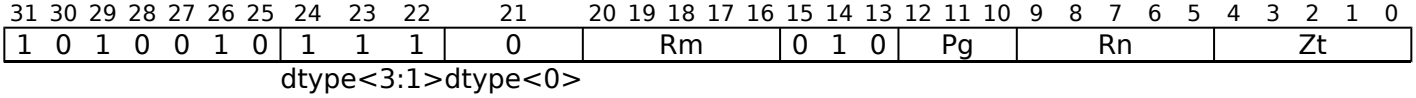
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1SB (scalar plus scalar)

Contiguous load signed bytes to vector (scalar index)

Contiguous load of signed bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.
It has encodings from 3 classes: 16-bit element , 32-bit element and 64-bit element

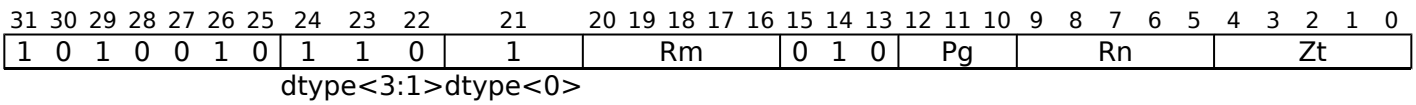
16-bit element



LD1SB { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 8;
boolean unsigned = FALSE;
```

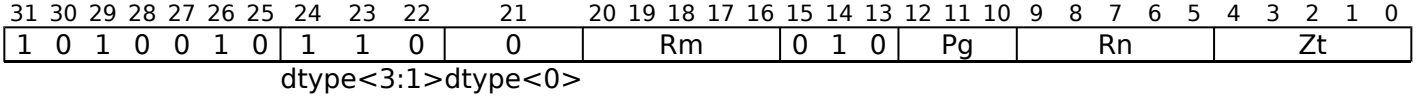
32-bit element



LD1SB { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = FALSE;
```

64-bit element



LD1SB { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

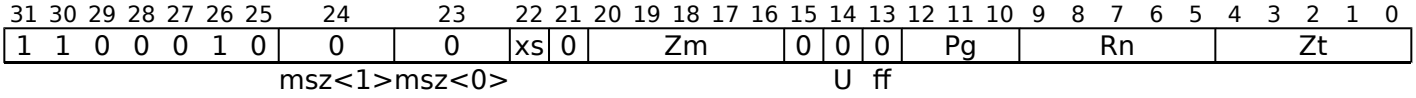
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1SB (scalar plus vector)

Gather load signed bytes to vector (vector index)

Gather load of signed bytes to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 3 classes: 32-bit unpacked unscaled offset , 32-bit unscaled offset and 64-bit unscaled offset

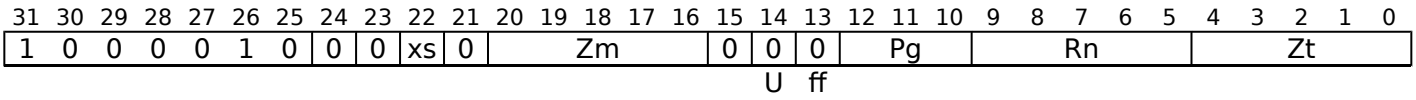
32-bit unpacked unscaled offset



LD1SB { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

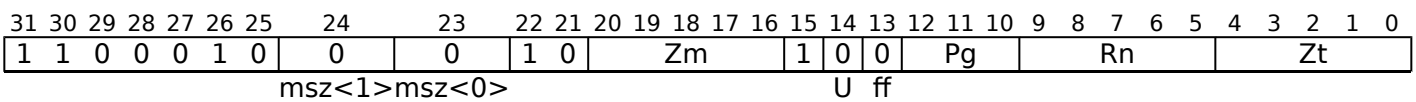
32-bit unscaled offset



LD1SB { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

64-bit unscaled offset



LD1SB { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

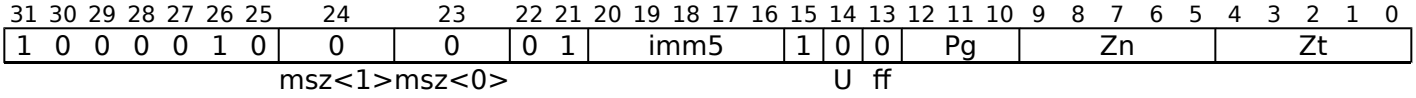
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

LD1SB (vector plus immediate)

Gather load signed bytes to vector (immediate index)

Gather load of signed bytes to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 2 classes: 32-bit element and 64-bit element

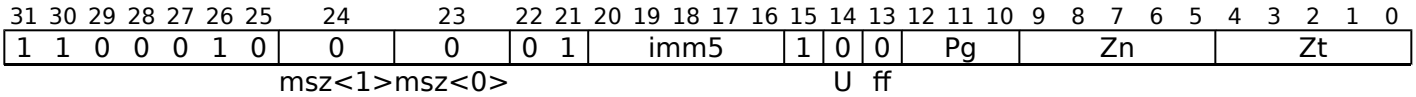
32-bit element



LD1SB { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm5);
```

64-bit element



LD1SB { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm5);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm>	Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

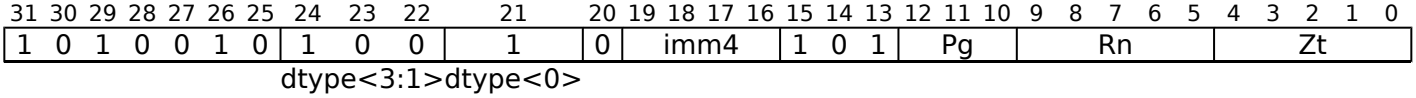
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1SH (scalar plus immediate)

Contiguous load signed halfwords to vector (immediate index)

Contiguous load of signed halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.
It has encodings from 2 classes: 32-bit element and 64-bit element

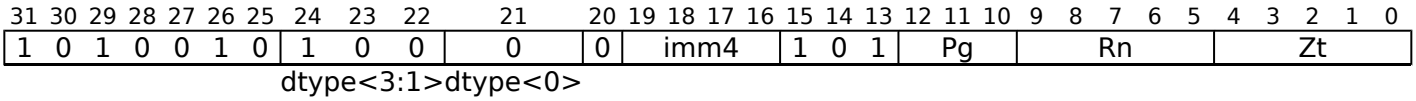
32-bit element



LD1SH { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
```

64-bit element



LD1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

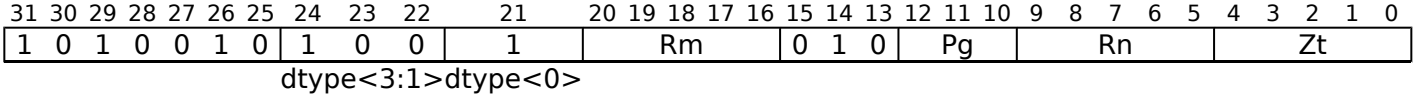
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1SH (scalar plus scalar)

Contiguous load signed halfwords to vector (scalar index)

Contiguous load of signed halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.
It has encodings from 2 classes: 32-bit element and 64-bit element

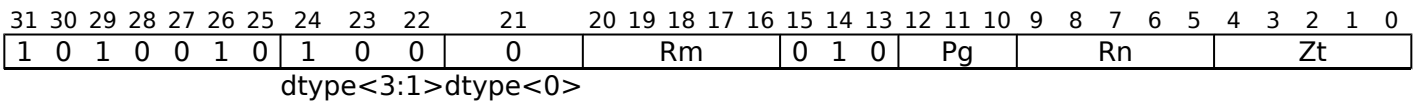
32-bit element



LD1SH { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = FALSE;
```

64-bit element



LD1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1SH (scalar plus vector)

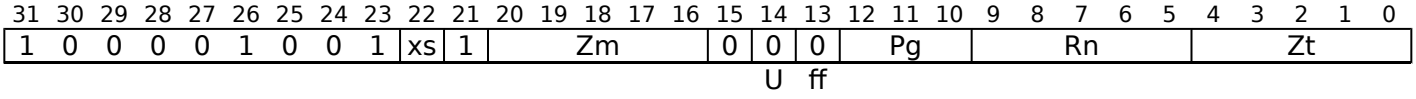
Gather load signed halfwords to vector (vector index)

Gather load of signed halfwords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

It has encodings from 6 classes: 32-bit scaled offset , 32-bit unpacked scaled offset , 32-bit unpacked unscaled offset , 32-bit unscaled offset , 64-bit scaled offset and 64-bit unscaled offset

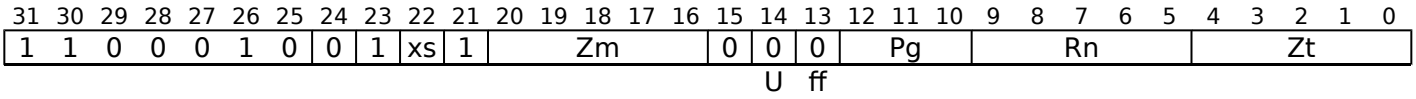
32-bit scaled offset



LD1SH { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #1]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```

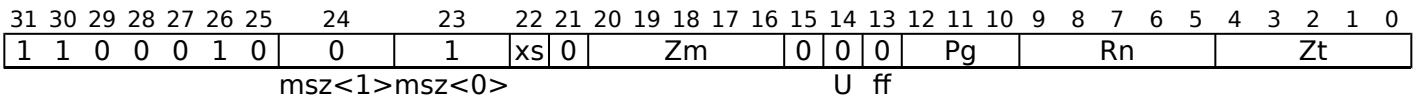
32-bit unpacked scaled offset



LD1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #1]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```

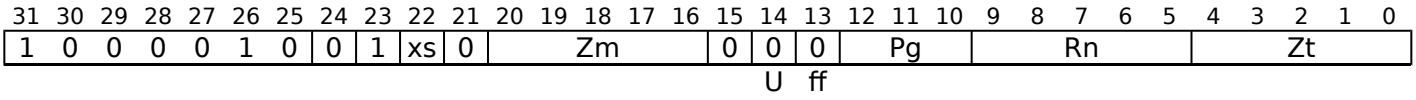
32-bit unpacked unscaled offset



LD1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

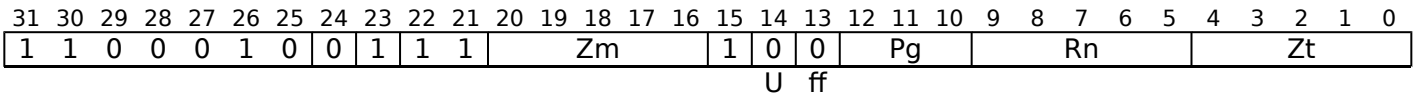
32-bit unscaled offset



LD1SH { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

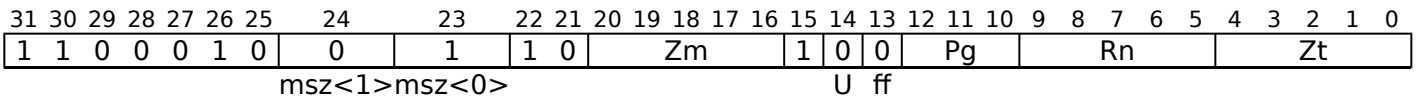
64-bit scaled offset



LD1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #1]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 1;
```

64-bit unscaled offset



LD1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm>	Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod>	Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

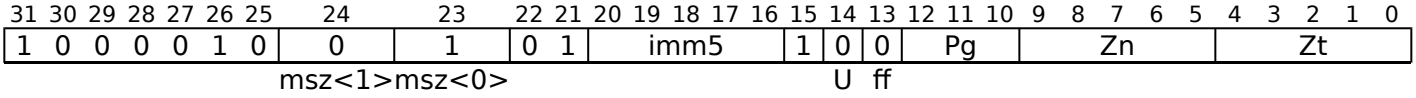
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1SH (vector plus immediate)

Gather load signed halfwords to vector (immediate index)

Gather load of signed halfwords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 2 classes: 32-bit element and 64-bit element

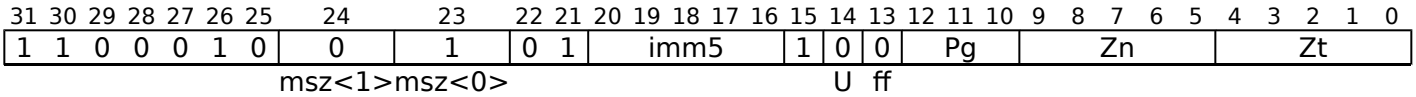
32-bit element



LD1SH { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm5);
```

64-bit element



LD1SH { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm5);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm>	Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the "imm5" field.

Operation

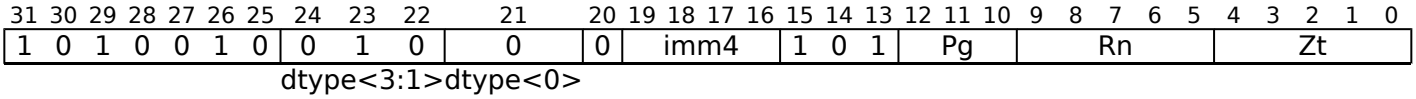
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1SW (scalar plus immediate)

Contiguous load signed words to vector (immediate index)

Contiguous load of signed words to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.



LD1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

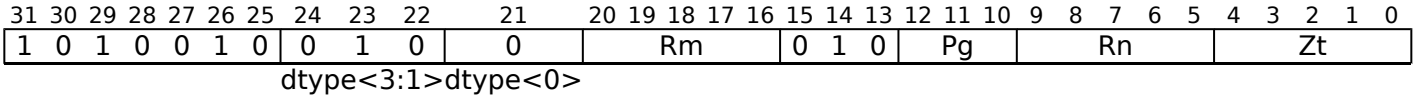
Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

LD1SW (scalar plus scalar)

Contiguous load signed words to vector (scalar index)

Contiguous load of signed words to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.



LD1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1SW (scalar plus vector)

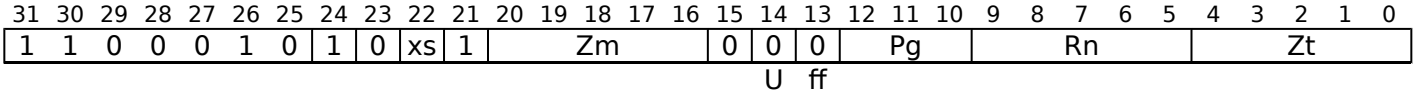
Gather load signed words to vector (vector index)

Gather load of signed words to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

It has encodings from 4 classes: 32-bit unpacked scaled offset , 32-bit unpacked unscaled offset , 64-bit scaled offset and 64-bit unscaled offset

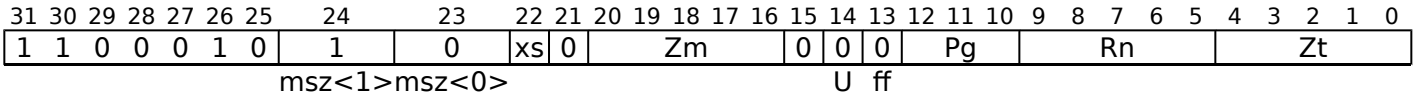
32-bit unpacked scaled offset



LD1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #2]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 2;
```

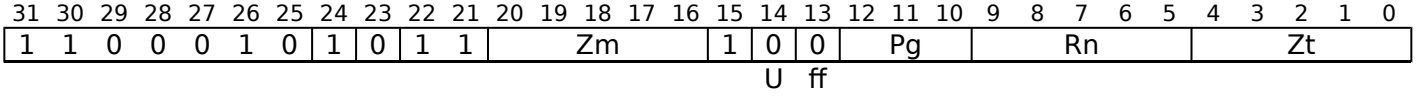
32-bit unpacked unscaled offset



LD1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

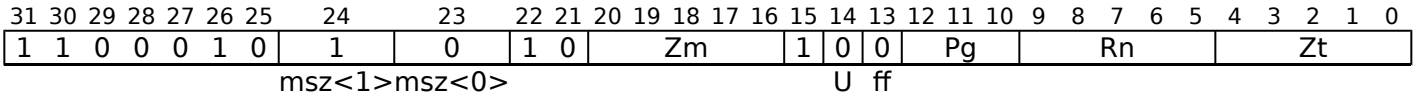
64-bit scaled offset



LD1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #2]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 2;
```

64-bit unscaled offset



LD1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

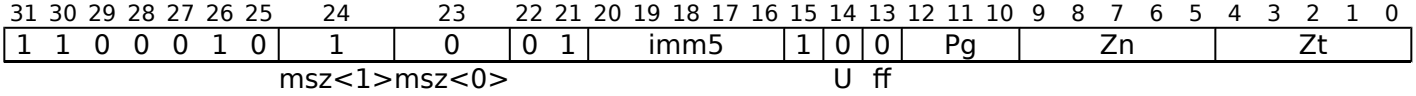
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1SW (vector plus immediate)

Gather load signed words to vector (immediate index)

Gather load of signed words to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.



LD1SW { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = FALSE;
integer offset = UInt(imm5);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm>	Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the "imm5" field.

Operation

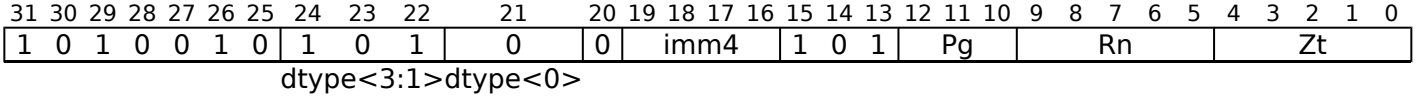
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

LD1W (scalar plus immediate)

Contiguous load unsigned words to vector (immediate index)

Contiguous load of unsigned words to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.
It has encodings from 2 classes: 32-bit element and 64-bit element

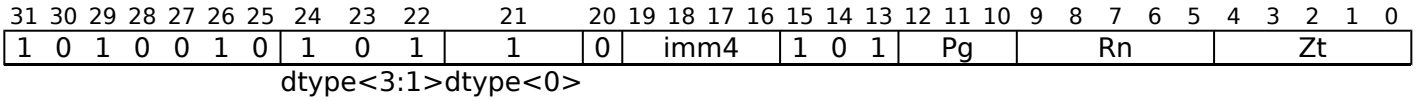
32-bit element



LD1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

64-bit element



LD1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

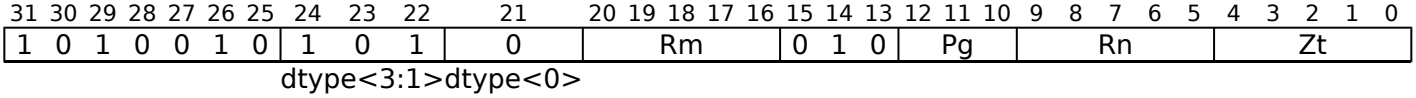
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1W (scalar plus scalar)

Contiguous load unsigned words to vector (scalar index)

Contiguous load of unsigned words to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.
It has encodings from 2 classes: 32-bit element and 64-bit element

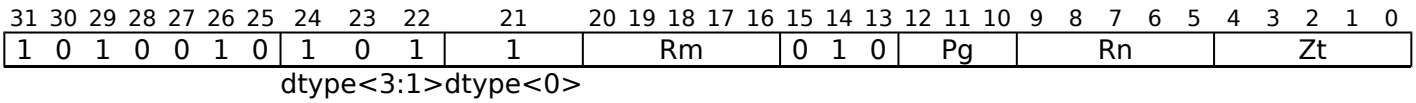
32-bit element



LD1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
boolean unsigned = TRUE;
```

64-bit element



LD1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1W (scalar plus vector)

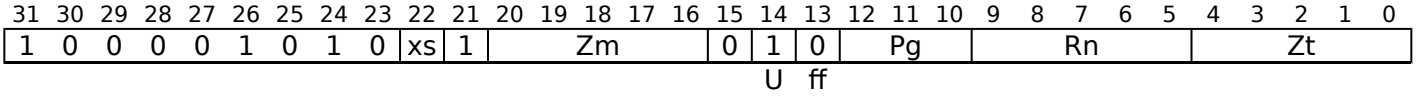
Gather load unsigned words to vector (vector index)

Gather load of unsigned words to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

It has encodings from 6 classes: 32-bit scaled offset , 32-bit unpacked scaled offset , 32-bit unpacked unscaled offset , 32-bit unscaled offset , 64-bit scaled offset and 64-bit unscaled offset

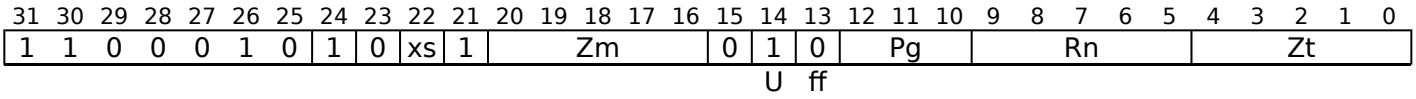
32-bit scaled offset



LD1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #2]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 2;
```

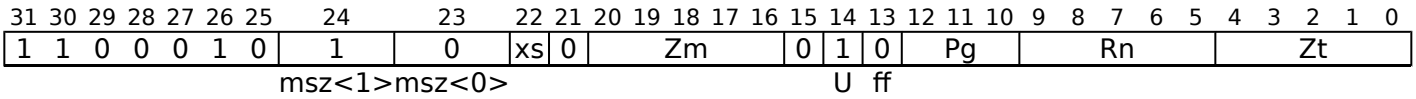
32-bit unpacked scaled offset



LD1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #2]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 2;
```

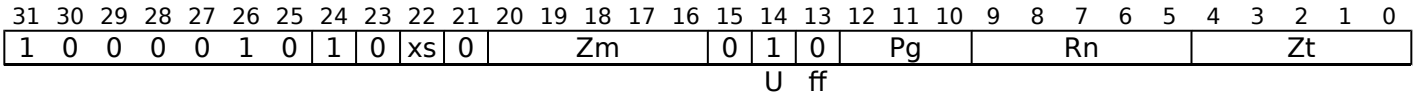
32-bit unpacked unscaled offset



LD1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

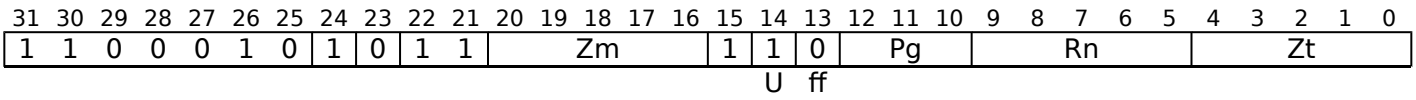
32-bit unscaled offset



LD1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

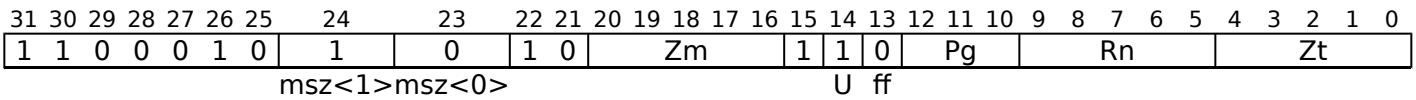
64-bit scaled offset



LD1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #2]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 2;
```

64-bit unscaled offset



LD1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

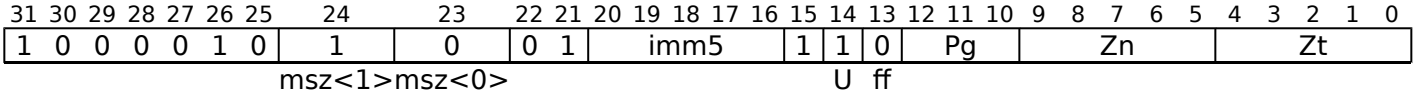
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

LD1W (vector plus immediate)

Gather load unsigned words to vector (immediate index)

Gather load of unsigned words to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 2 classes: 32-bit element and 64-bit element

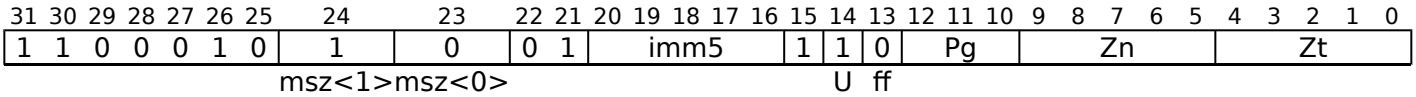
32-bit element



LD1W { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

64-bit element



LD1W { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm>	Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the "imm5" field.

Operation

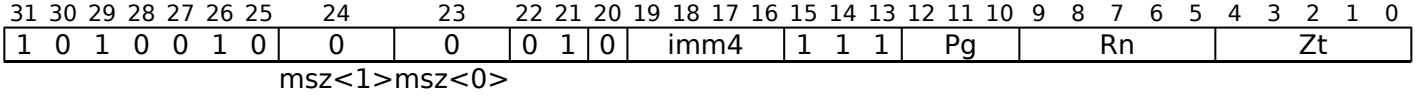
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        data = Mem[addr, mbytes, AccType_SVE];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD2B (scalar plus immediate)

Contiguous load two-byte structures to two vectors (immediate index)

Contiguous load two-byte structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication,
Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.



LD2B { <Zt1>.B, <Zt2>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
integer offset = SInt(imm4);
constant integer nreg = 2;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

Operation

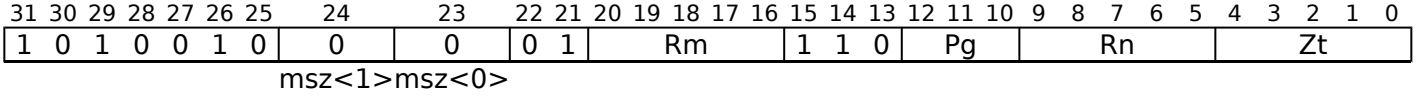
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD2B (scalar plus scalar)

Contiguous load two-byte structures to two vectors (scalar index)

Contiguous load two-byte structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.
Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.



LD2B { <Zt1>.B, <Zt2>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
constant integer nreg = 2;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

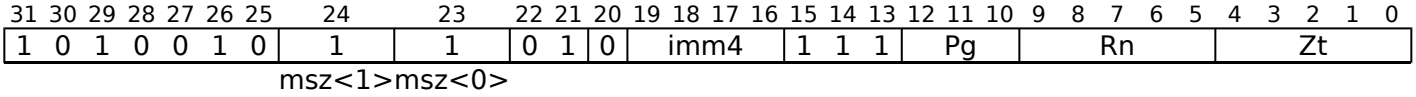
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD2D (scalar plus immediate)

Contiguous load two-doubleword structures to two vectors (immediate index)

Contiguous load two-doubleword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication,
Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.



LD2D { <Zt1>.D, <Zt2>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
integer offset = SInt(imm4);
constant integer nreg = 2;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

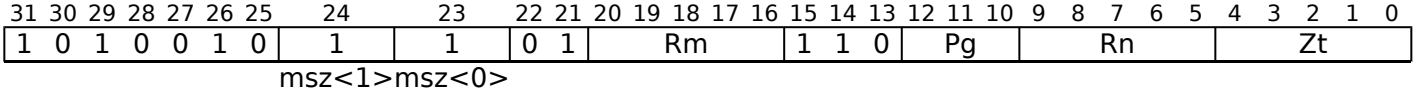
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD2D (scalar plus scalar)

Contiguous load two-doubleword structures to two vectors (scalar index)

Contiguous load two-doubleword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.



LD2D { <Zt1>.D, <Zt2>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer nreg = 2;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

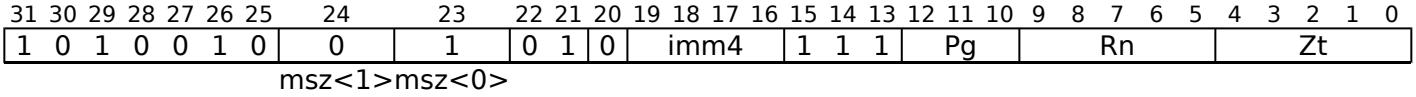
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD2H (scalar plus immediate)

Contiguous load two-halfword structures to two vectors (immediate index)

Contiguous load two-halfword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication,
Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.



LD2H { <Zt1>.H, <Zt2>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
integer offset = SInt(imm4);
constant integer nreg = 2;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

Operation

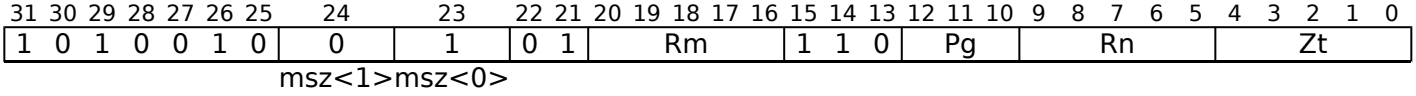
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD2H (scalar plus scalar)

Contiguous load two-halfword structures to two vectors (scalar index)

Contiguous load two-halfword structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.
Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.



LD2H { <Zt1>.H, <Zt2>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer nreg = 2;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

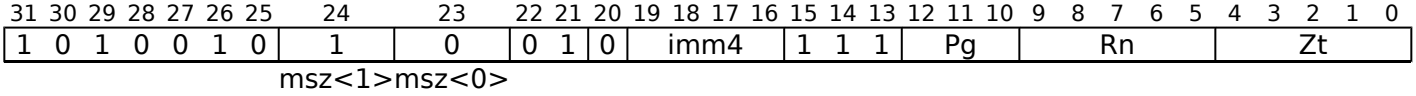
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD2W (scalar plus immediate)

Contiguous load two-word structures to two vectors (immediate index)

Contiguous load two-word structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication,
Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.



LD2W { <Zt1>.S, <Zt2>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
integer offset = SInt(imm4);
constant integer nreg = 2;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

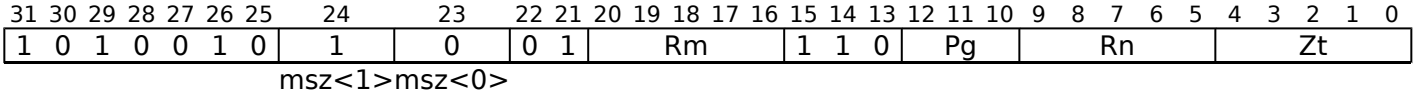
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD2W (scalar plus scalar)

Contiguous load two-word structures to two vectors (scalar index)

Contiguous load two-word structures, each to the same element number in two vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.

Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the two destination vector registers.



LD2W { <Zt1>.S, <Zt2>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer nreg = 2;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

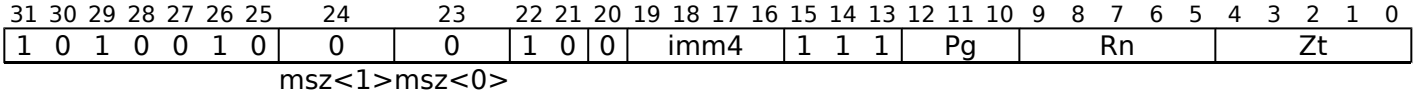
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD3B (scalar plus immediate)

Contiguous load three-byte structures to three vectors (immediate index)

Contiguous load three-byte structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication,
Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.



LD3B { <Zt1>.B, <Zt2>.B, <Zt3>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
integer offset = SInt(imm4);
constant integer nreg = 3;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

Operation

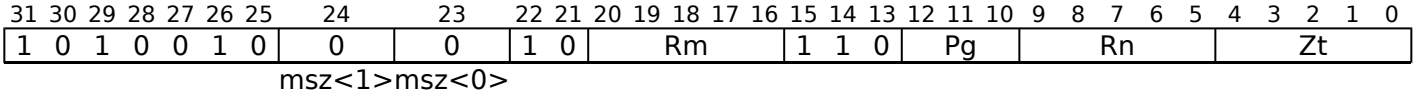
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD3B (scalar plus scalar)

Contiguous load three-byte structures to three vectors (scalar index)

Contiguous load three-byte structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.
Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.



LD3B { <Zt1>.B, <Zt2>.B, <Zt3>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
constant integer nreg = 3;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

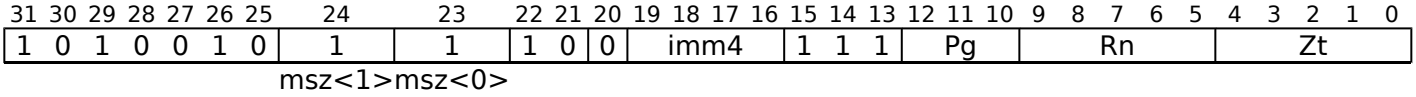
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD3D (scalar plus immediate)

Contiguous load three-doubleword structures to three vectors (immediate index)

Contiguous load three-doubleword structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication,
Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.



LD3D { <Zt1>.D, <Zt2>.D, <Zt3>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
integer offset = SInt(imm4);
constant integer nreg = 3;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

Operation

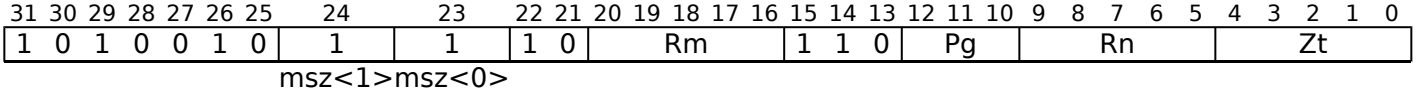
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD3D (scalar plus scalar)

Contiguous load three-doubleword structures to three vectors (scalar index)

Contiguous load three-doubleword structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.
Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.



LD3D { <Zt1>.D, <Zt2>.D, <Zt3>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer nreg = 3;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

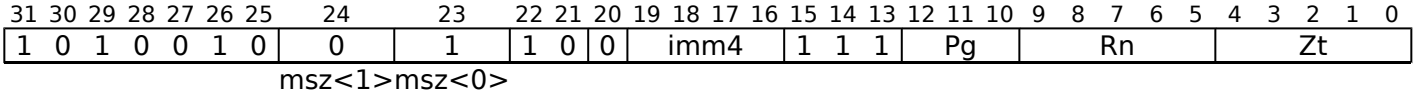
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD3H (scalar plus immediate)

Contiguous load three-halfword structures to three vectors (immediate index)

Contiguous load three-halfword structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication,
Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.



LD3H { <Zt1>.H, <Zt2>.H, <Zt3>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
integer offset = SInt(imm4);
constant integer nreg = 3;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

Operation

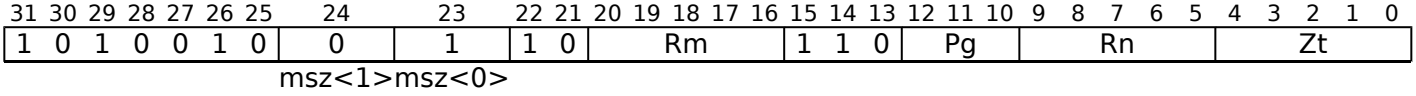
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD3H (scalar plus scalar)

Contiguous load three-halfword structures to three vectors (scalar index)

Contiguous load three-halfword structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.
Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.



LD3H { <Zt1>.H, <Zt2>.H, <Zt3>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer nreg = 3;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

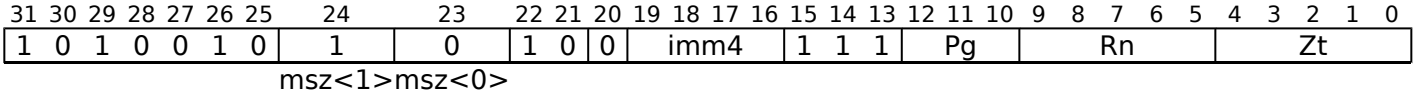
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD3W (scalar plus immediate)

Contiguous load three-word structures to three vectors (immediate index)

Contiguous load three-word structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication,
Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.



LD3W { <Zt1>.S, <Zt2>.S, <Zt3>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
integer offset = SInt(imm4);
constant integer nreg = 3;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

Operation

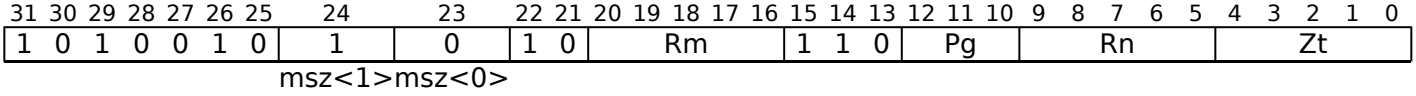
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD3W (scalar plus scalar)

Contiguous load three-word structures to three vectors (scalar index)

Contiguous load three-word structures, each to the same element number in three vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.
Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the three destination vector registers.



LD3W { <Zt1>.S, <Zt2>.S, <Zt3>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer nreg = 3;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

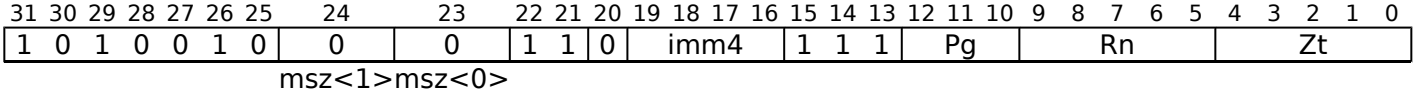
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD4B (scalar plus immediate)

Contiguous load four-byte structures to four vectors (immediate index)

Contiguous load four-byte structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication,
Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.



LD4B { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
integer offset = SInt(imm4);
constant integer nreg = 4;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4>	Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

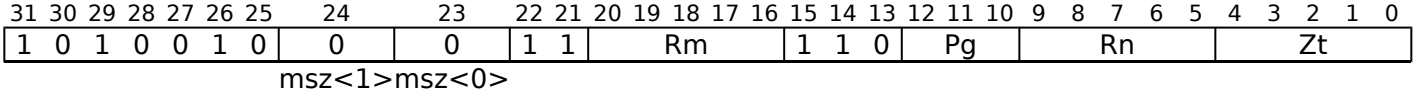
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD4B (scalar plus scalar)

Contiguous load four-byte structures to four vectors (scalar index)

Contiguous load four-byte structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.
Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive bytes in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.



LD4B { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
constant integer nreg = 4;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4>	Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

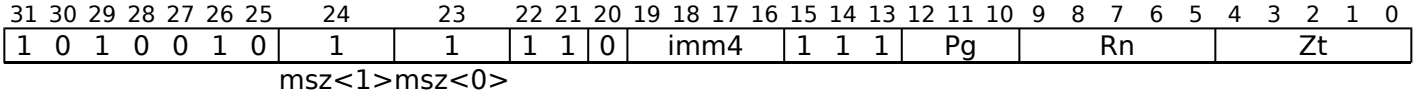
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD4D (scalar plus immediate)

Contiguous load four-doubleword structures to four vectors (immediate index)

Contiguous load four-doubleword structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication,
Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.



LD4D { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
integer offset = SInt(imm4);
constant integer nreg = 4;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4>	Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

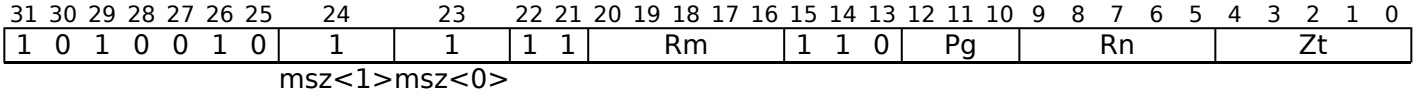
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD4D (scalar plus scalar)

Contiguous load four-doubleword structures to four vectors (scalar index)

Contiguous load four-doubleword structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.
Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive doublewords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.



LD4D { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer nreg = 4;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4>	Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

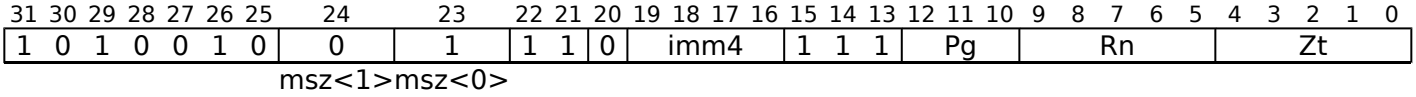
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD4H (scalar plus immediate)

Contiguous load four-halfword structures to four vectors (immediate index)

Contiguous load four-halfword structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication,
Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.



LD4H { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
integer offset = SInt(imm4);
constant integer nreg = 4;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4>	Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

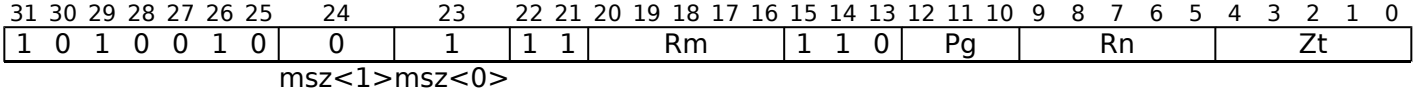
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD4H (scalar plus scalar)

Contiguous load four-halfword structures to four vectors (scalar index)

Contiguous load four-halfword structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.
Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive halfwords in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.



LD4H { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer nreg = 4;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4>	Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

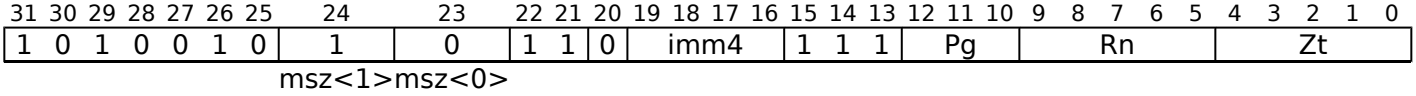
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD4W (scalar plus immediate)

Contiguous load four-word structures to four vectors (immediate index)

Contiguous load four-word structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication,
Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.



LD4W { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
integer offset = SInt(imm4);
constant integer nreg = 4;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4>	Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

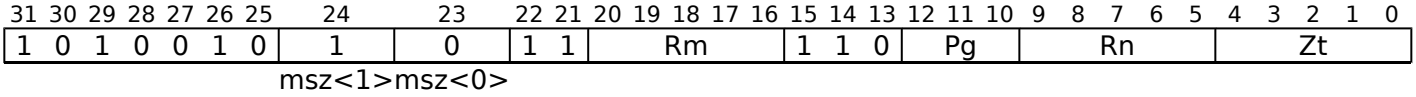
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD4W (scalar plus scalar)

Contiguous load four-word structures to four vectors (scalar index)

Contiguous load four-word structures, each to the same element number in four vector registers from the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.
Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive words in memory which make up each structure. Inactive elements will not cause a read from Device memory or signal a fault, and the corresponding element is set to zero in each of the four destination vector registers.



LD4W { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer nreg = 4;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4>	Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Elem[values[r], e, esize] = Mem[addr, mbytes, AccType_SVE];
        else
            Elem[values[r], e, esize] = Zeros(esize);

for r = 0 to nreg-1
    Z[(t+r) MOD 32, VL] = values[r];
```

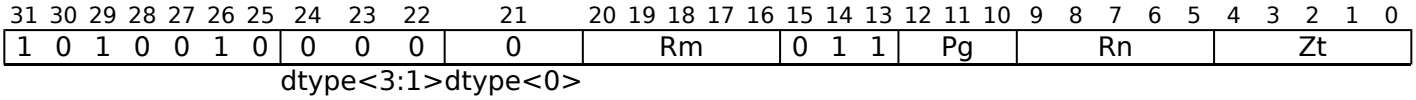
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDFF1B (scalar plus scalar)

Contiguous load first-fault unsigned bytes to vector (scalar index)

Contiguous load with first-faulting behavior of unsigned bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 4 classes: 8-bit element , 16-bit element , 32-bit element and 64-bit element

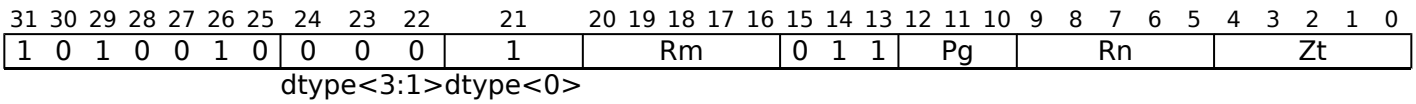
8-bit element



LDFF1B { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, <Xm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
constant integer msize = 8;
boolean unsigned = TRUE;
```

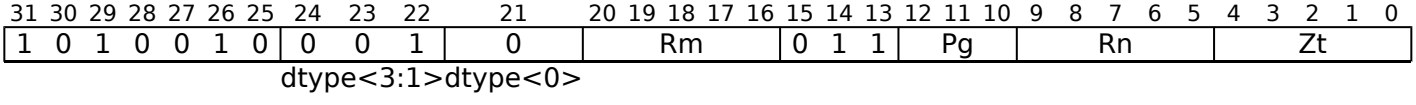
16-bit element



LDFF1B { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, <Xm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 8;
boolean unsigned = TRUE;
```

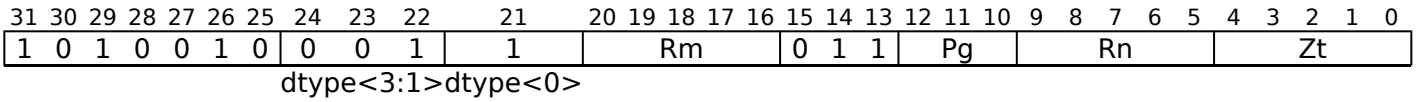
32-bit element



LDFF1B { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, <Xm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = TRUE;
```

64-bit element



LDFF1B { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDFF1B (scalar plus vector)

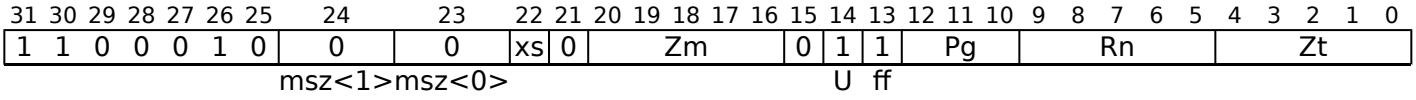
Gather load first-fault unsigned bytes to vector (vector index)

Gather load with first-faulting behavior of unsigned bytes to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

It has encodings from 3 classes: 32-bit unpacked unscaled offset , 32-bit unscaled offset and 64-bit unscaled offset

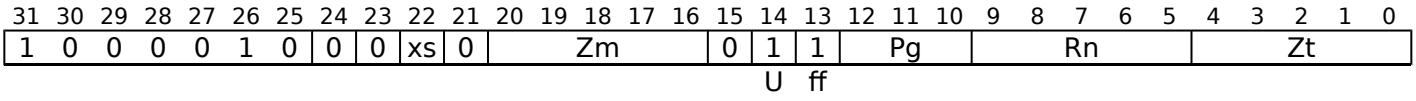
32-bit unpacked unscaled offset



LDFF1B { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

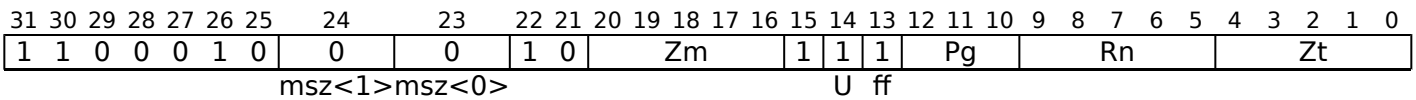
32-bit unscaled offset



LDFF1B { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

64-bit unscaled offset



LDFF1B { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

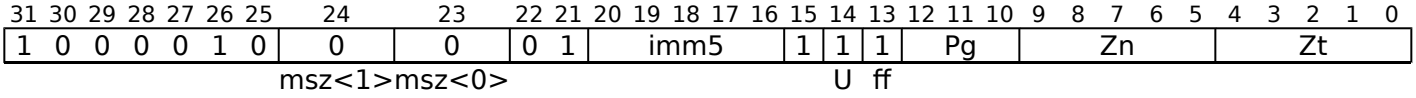
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(VL) offset;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

LDFF1B (vector plus immediate)

Gather load first-fault unsigned bytes to vector (immediate index)

Gather load with first-faulting behavior of unsigned bytes to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 2 classes: 32-bit element and 64-bit element

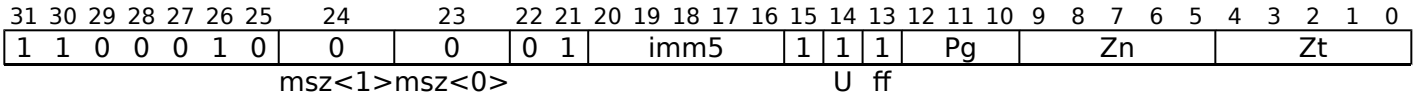
32-bit element



LDFF1B { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

64-bit element



LDFF1B { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm>	Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

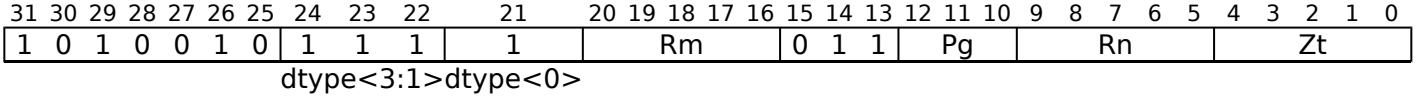
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDFF1D (scalar plus scalar)

Contiguous load first-fault doublewords to vector (scalar index)

Contiguous load with first-faulting behavior of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.



LDFF1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #3}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDFF1D (scalar plus vector)

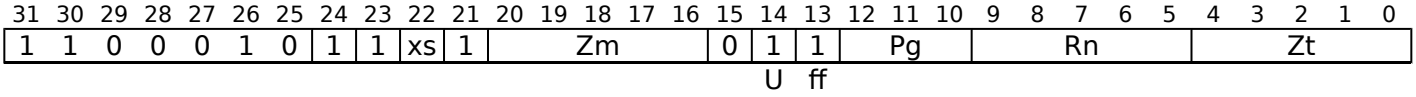
Gather load first-fault doublewords to vector (vector index)

Gather load with first-faulting behavior of doublewords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 8. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

It has encodings from 4 classes: 32-bit unpacked scaled offset , 32-bit unpacked unscaled offset , 64-bit scaled offset and 64-bit unscaled offset

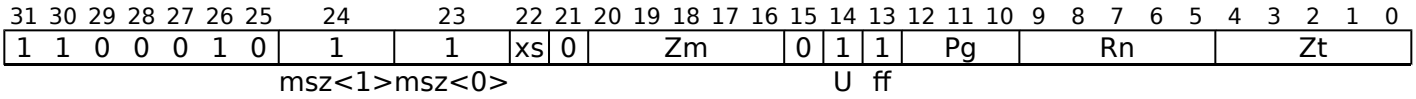
32-bit unpacked scaled offset



LDFF1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #3]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 3;
```

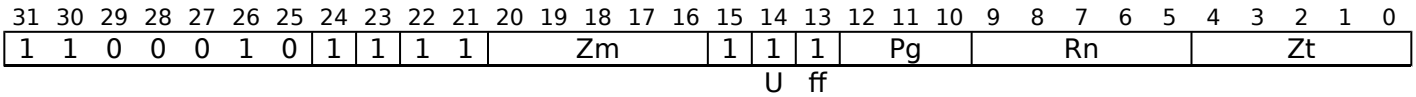
32-bit unpacked unscaled offset



LDFF1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

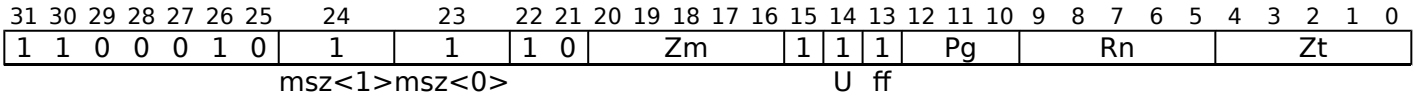
64-bit scaled offset



LDFF1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #3]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 3;
```

64-bit unscaled offset



LDFF1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

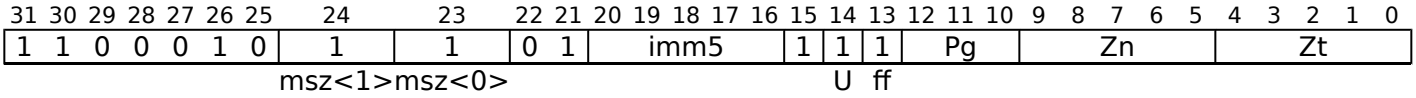
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(VL) offset;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

LDFF1D (vector plus immediate)

Gather load first-fault doublewords to vector (immediate index)

Gather load with first-faulting behavior of doublewords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 8 in the range 0 to 248. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.



LDFF1D { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm>	Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 248, defaulting to 0, encoded in the "imm5" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

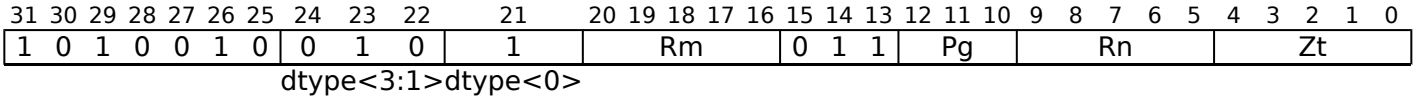
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDFF1H (scalar plus scalar)

Contiguous load first-fault unsigned halfwords to vector (scalar index)

Contiguous load with first-faulting behavior of unsigned halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 3 classes: 16-bit element , 32-bit element and 64-bit element

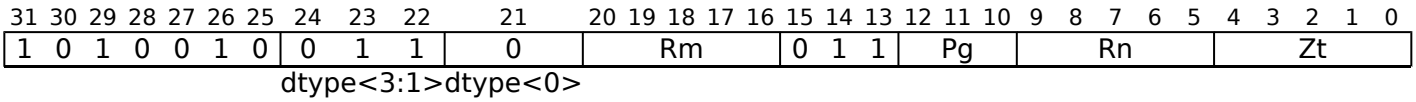
16-bit element



LDFF1H { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #1}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 16;
boolean unsigned = TRUE;
```

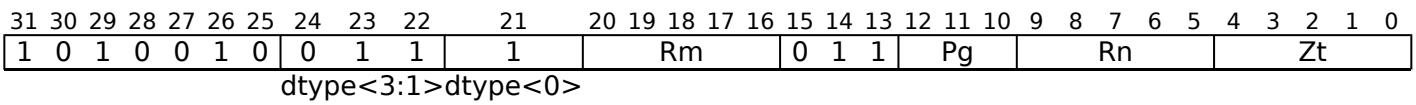
32-bit element



LDFF1H { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #1}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = TRUE;
```

64-bit element



LDFF1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #1}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDFF1H (scalar plus vector)

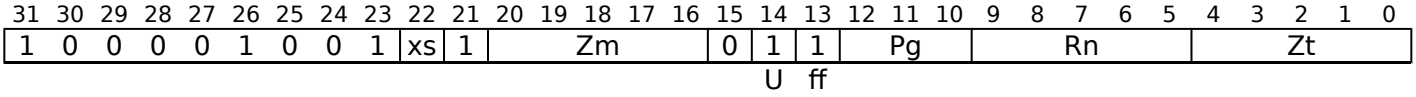
Gather load first-fault unsigned halfwords to vector (vector index)

Gather load with first-faulting behavior of unsigned halfwords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

It has encodings from 6 classes: 32-bit scaled offset , 32-bit unpacked scaled offset , 32-bit unpacked unscaled offset , 32-bit unscaled offset , 64-bit scaled offset and 64-bit unscaled offset

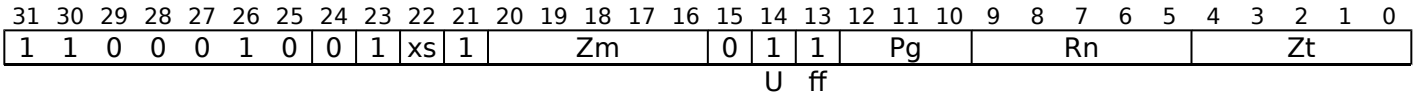
32-bit scaled offset



LDFF1H { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #1]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```

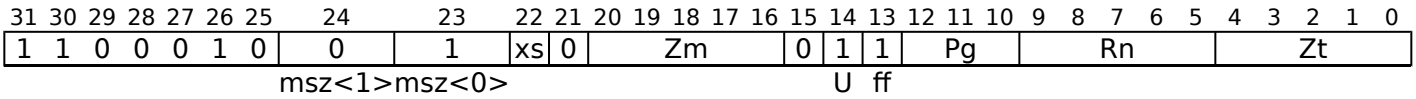
32-bit unpacked scaled offset



LDFF1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #1]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```

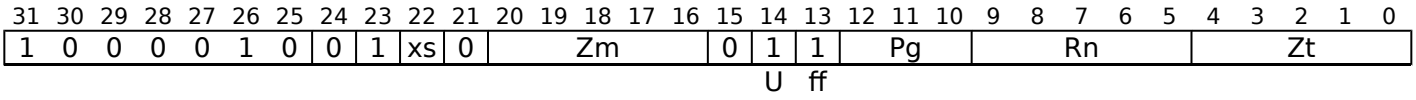
32-bit unpacked unscaled offset



LDFF1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

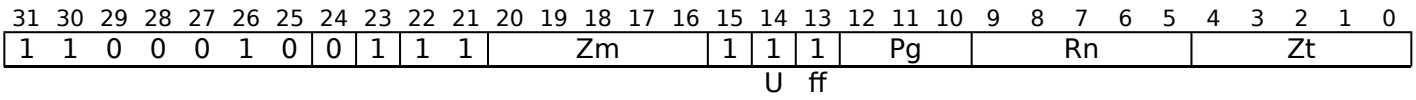
32-bit unscaled offset



LDFF1H { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

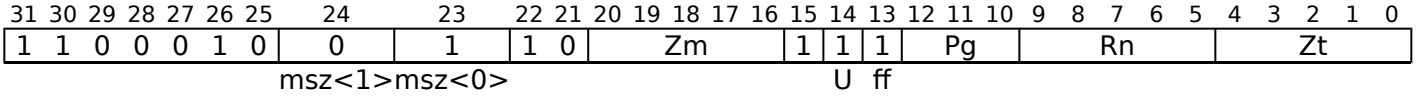
64-bit scaled offset



LDFF1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #1]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 1;
```

64-bit unscaled offset



LDFF1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

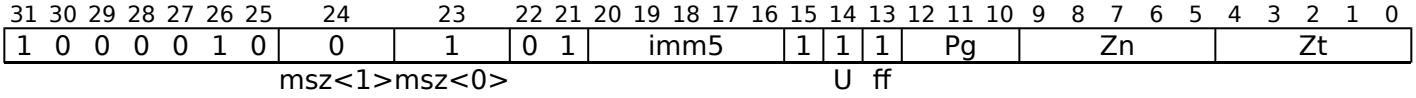
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(VL) offset;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

LDFF1H (vector plus immediate)

Gather load first-fault unsigned halfwords to vector (immediate index)

Gather load with first-faulting behavior of unsigned halfwords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 2 classes: 32-bit element and 64-bit element

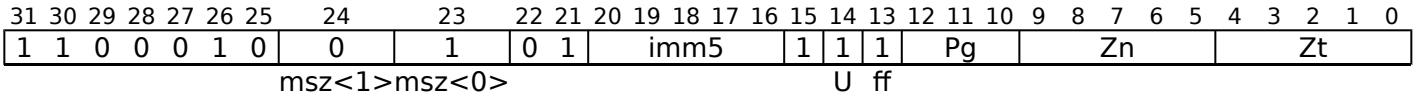
32-bit element



LDFF1H { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

64-bit element



LDFF1H { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm>	Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the "imm5" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

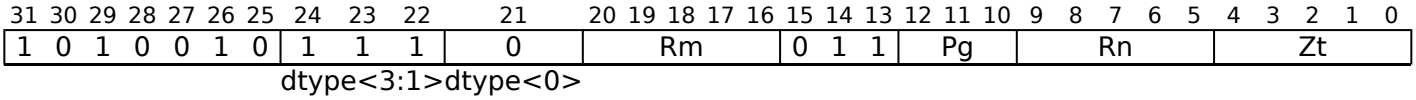
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDFF1SB (scalar plus scalar)

Contiguous load first-fault signed bytes to vector (scalar index)

Contiguous load with first-faulting behavior of signed bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 3 classes: 16-bit element , 32-bit element and 64-bit element

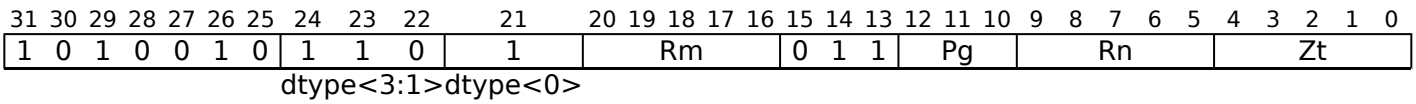
16-bit element



LDFF1SB { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, <Xm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 8;
boolean unsigned = FALSE;
```

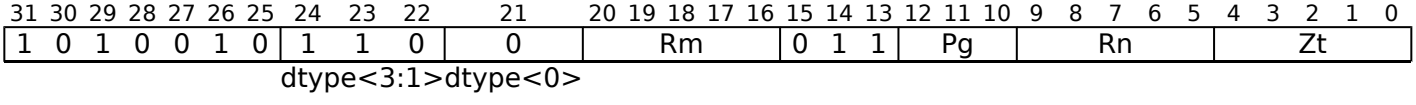
32-bit element



LDFF1SB { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, <Xm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = FALSE;
```

64-bit element



LDFF1SB { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

LDFF1SB (scalar plus vector)

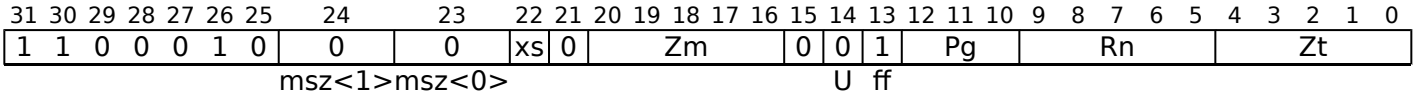
Gather load first-fault signed bytes to vector (vector index)

Gather load with first-faulting behavior of signed bytes to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

It has encodings from 3 classes: 32-bit unpacked unscaled offset , 32-bit unscaled offset and 64-bit unscaled offset

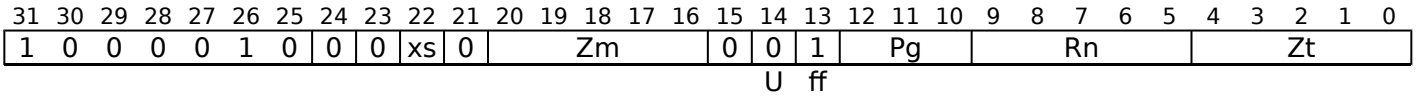
32-bit unpacked unscaled offset



LDFF1SB { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

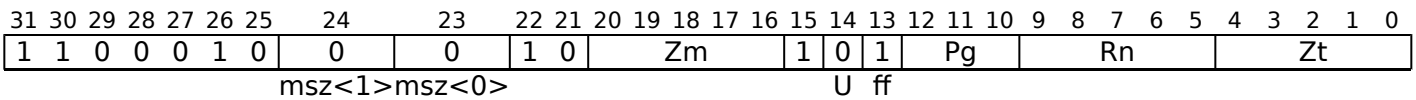
32-bit unscaled offset



LDFF1SB { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

64-bit unscaled offset



LDFF1SB { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

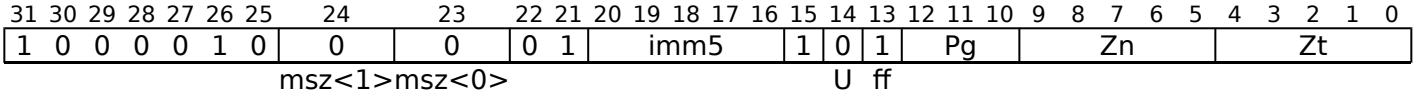
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(VL) offset;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

LDFF1SB (vector plus immediate)

Gather load first-fault signed bytes to vector (immediate index)

Gather load with first-faulting behavior of signed bytes to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 2 classes: 32-bit element and 64-bit element

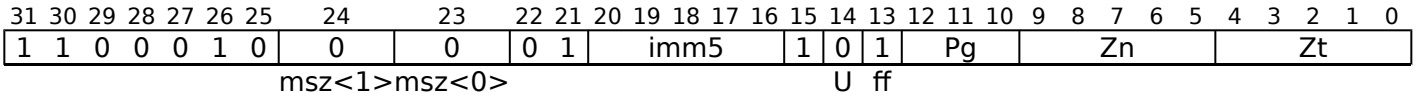
32-bit element



LDFF1SB { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm5);
```

64-bit element



LDFF1SB { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = UInt(imm5);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm>	Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

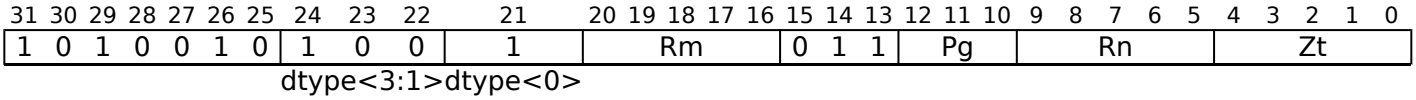
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDFF1SH (scalar plus scalar)

Contiguous load first-fault signed halfwords to vector (scalar index)

Contiguous load with first-faulting behavior of signed halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 2 classes: 32-bit element and 64-bit element

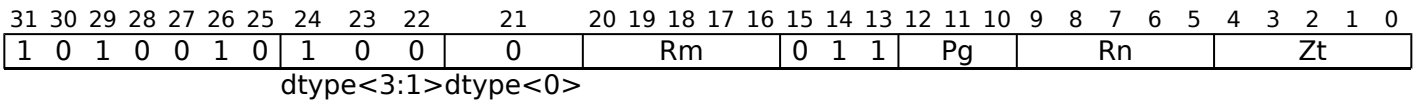
32-bit element



LDFF1SH { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #1}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = FALSE;
```

64-bit element



LDFF1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #1}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

LDFF1SH (scalar plus vector)

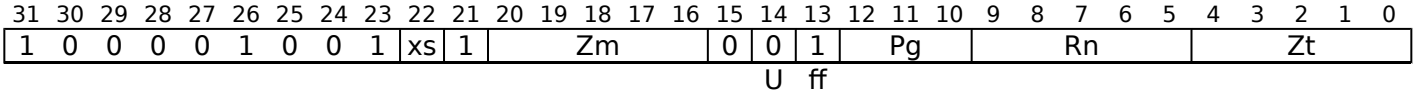
Gather load first-fault signed halfwords to vector (vector index)

Gather load with first-faulting behavior of signed halfwords to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

It has encodings from 6 classes: 32-bit scaled offset , 32-bit unpacked scaled offset , 32-bit unpacked unscaled offset , 32-bit unscaled offset , 64-bit scaled offset and 64-bit unscaled offset

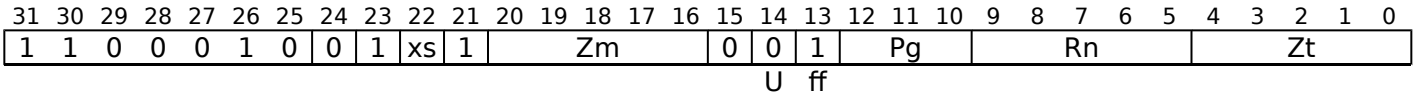
32-bit scaled offset



LDFF1SH { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #1]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```

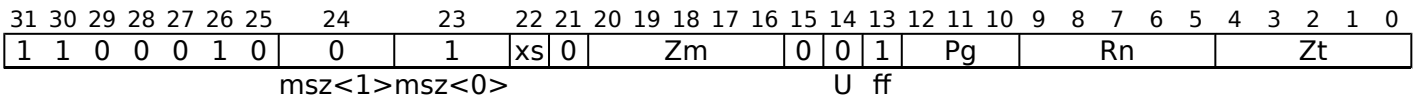
32-bit unpacked scaled offset



LDFF1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #1]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```

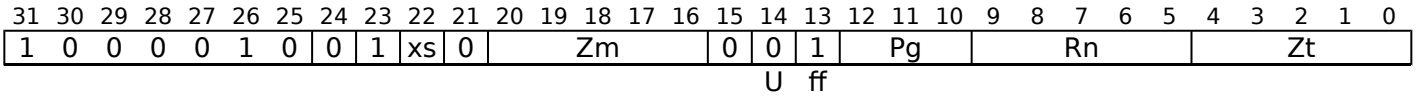
32-bit unpacked unscaled offset



LDFF1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

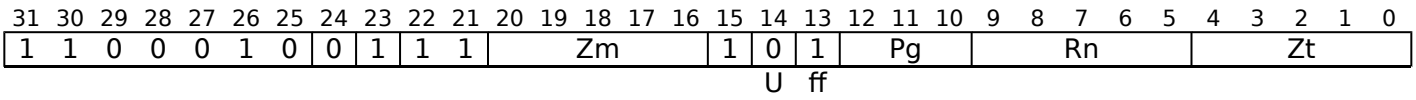
32-bit unscaled offset



LDFF1SH { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

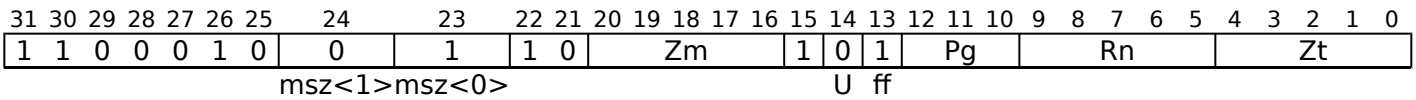
64-bit scaled offset



LDFF1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #1]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 1;
```

64-bit unscaled offset



LDFF1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

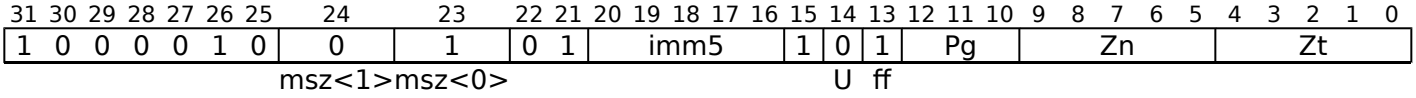
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(VL) offset;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

LDFF1SH (vector plus immediate)

Gather load first-fault signed halfwords to vector (immediate index)

Gather load with first-faulting behavior of signed halfwords to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 2 classes: 32-bit element and 64-bit element

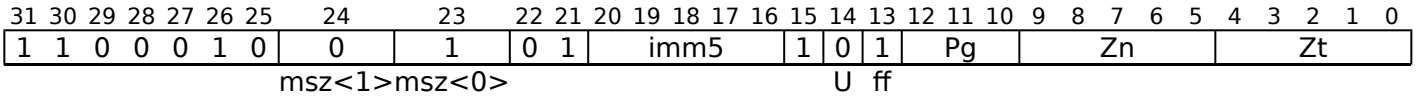
32-bit element



LDFF1SH { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm5);
```

64-bit element



LDFF1SH { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = FALSE;
integer offset = UInt(imm5);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm>	Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the "imm5" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

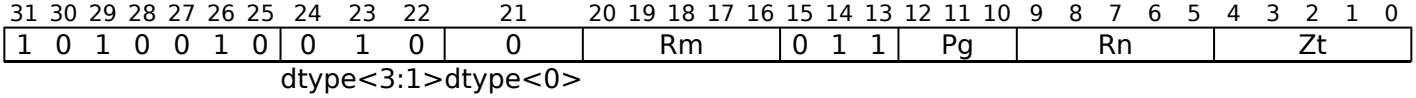
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDFF1SW (scalar plus scalar)

Contiguous load first-fault signed words to vector (scalar index)

Contiguous load with first-faulting behavior of signed words to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.



LDFF1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #2}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDFF1SW (scalar plus vector)

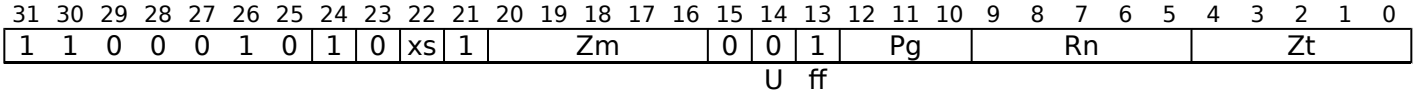
Gather load first-fault signed words to vector (vector index)

Gather load with first-faulting behavior of signed words to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

It has encodings from 4 classes: 32-bit unpacked scaled offset , 32-bit unpacked unscaled offset , 64-bit scaled offset and 64-bit unscaled offset

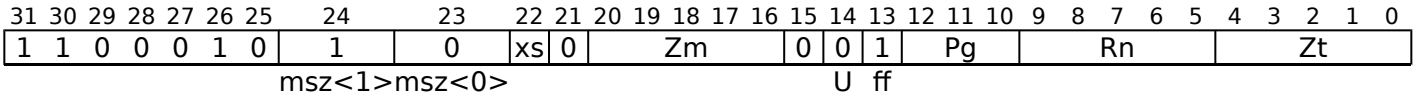
32-bit unpacked scaled offset



LDFF1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #2]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 2;
```

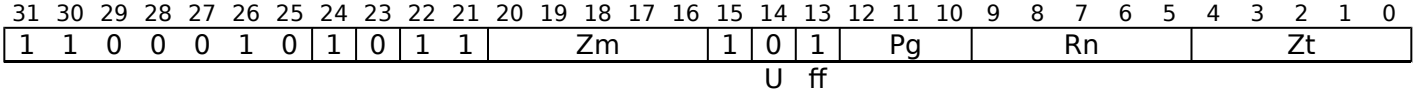
32-bit unpacked unscaled offset



LDFF1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 32;
boolean unsigned = FALSE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

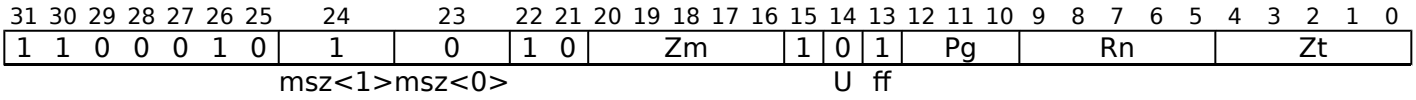
64-bit scaled offset



LDFF1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #2]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 2;
```

64-bit unscaled offset



LDFF1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 64;
boolean unsigned = FALSE;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

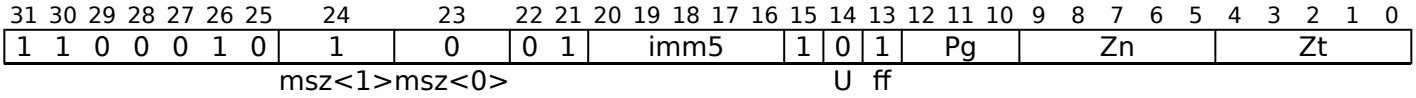
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(VL) offset;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

LDFF1SW (vector plus immediate)

Gather load first-fault signed words to vector (immediate index)

Gather load with first-faulting behavior of signed words to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.



LDFF1SW { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = FALSE;
integer offset = UInt(imm5);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm>	Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the "imm5" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

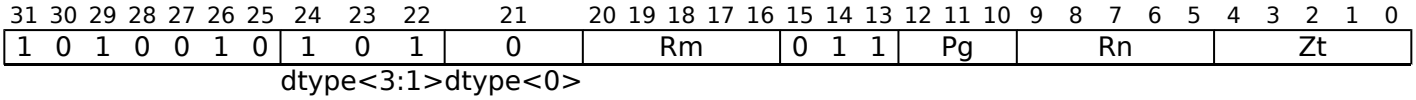
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDFF1W (scalar plus scalar)

Contiguous load first-fault unsigned words to vector (scalar index)

Contiguous load with first-faulting behavior of unsigned words to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 2 classes: 32-bit element and 64-bit element

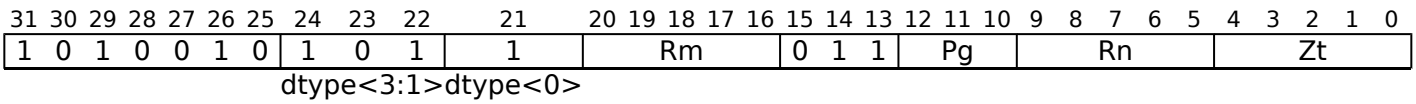
32-bit element



LDFF1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #2}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
boolean unsigned = TRUE;
```

64-bit element



LDFF1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #2}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
bits(64) offset;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_CNOTFIRST];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDFF1W (scalar plus vector)

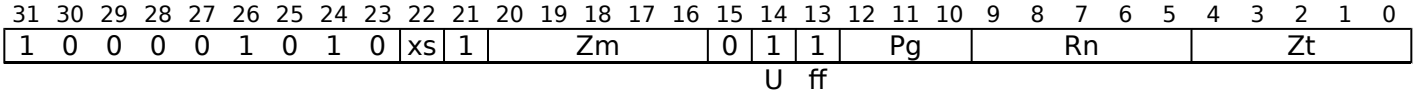
Gather load first-fault unsigned words to vector (vector index)

Gather load with first-faulting behavior of unsigned words to active elements of a vector register from memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

It has encodings from 6 classes: 32-bit scaled offset , 32-bit unpacked scaled offset , 32-bit unpacked unscaled offset , 32-bit unscaled offset , 64-bit scaled offset and 64-bit unscaled offset

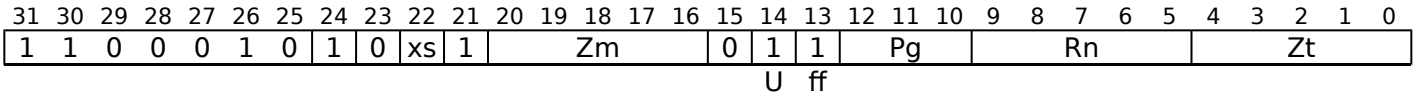
32-bit scaled offset



LDFF1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod> #2]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 2;
```

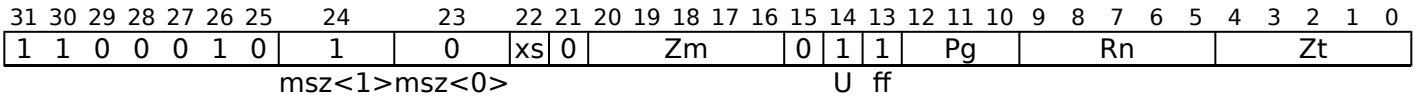
32-bit unpacked scaled offset



LDFF1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod> #2]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 2;
```

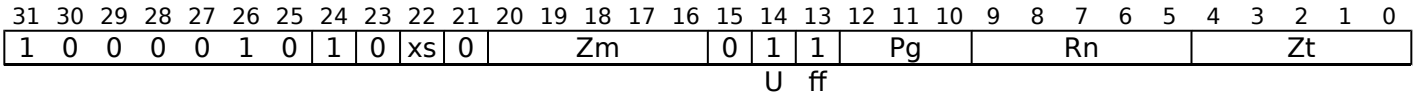
32-bit unpacked unscaled offset



LDFF1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

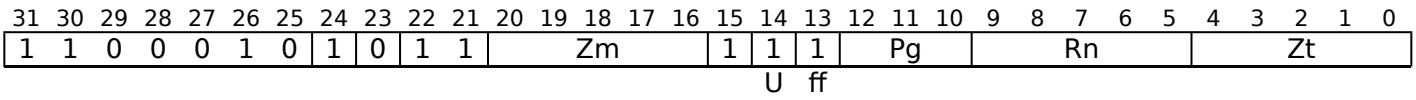
32-bit unscaled offset



LDFF1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Zm>.S, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
integer offs_size = 32;
boolean unsigned = TRUE;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

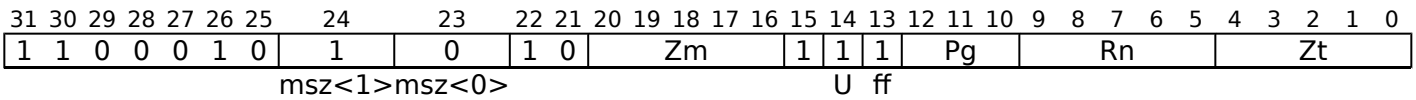
64-bit scaled offset



LDFF1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D, LSL #2]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 2;
```

64-bit unscaled offset



LDFF1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 64;
boolean unsigned = TRUE;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

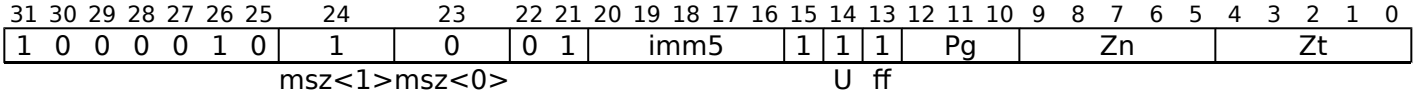
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(VL) offset;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

LDFF1W (vector plus immediate)

Gather load first-fault unsigned words to vector (immediate index)

Gather load with first-faulting behavior of unsigned words to active elements of a vector register from memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 2 classes: 32-bit element and 64-bit element

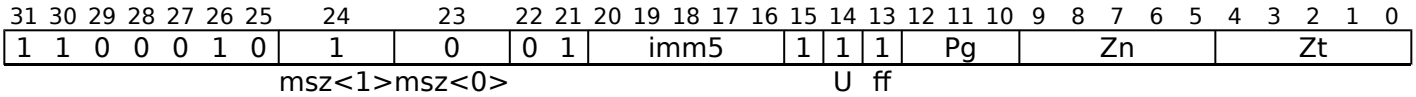
32-bit element



LDFF1W { <Zt>.S }, <Pg>/Z, [<Zn>.S{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

64-bit element



LDFF1W { <Zt>.D }, <Pg>/Z, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = TRUE;
integer offset = UInt(imm5);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm>	Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the "imm5" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean first = TRUE;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        if first then
            // Mem[] will not return if a fault is detected for the first active element
            data = Mem[addr, mbytes, AccType_SVE];
            first = FALSE;
        else
            // MemNF[] will return fault=TRUE if access is not performed for any reason
            (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

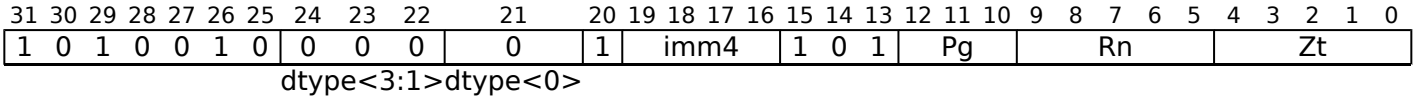
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDNF1B

Contiguous load non-fault unsigned bytes to vector (immediate index)

Contiguous load with non-faulting behavior of unsigned bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 4 classes: 8-bit element , 16-bit element , 32-bit element and 64-bit element

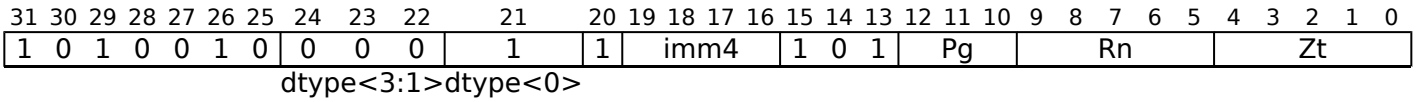
8-bit element



LDNF1B { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

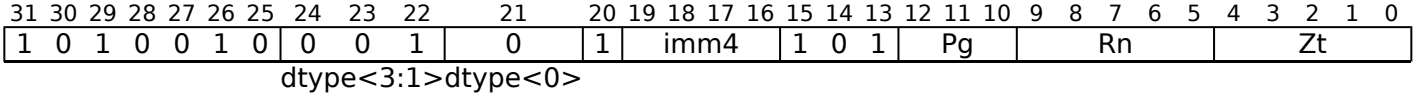
16-bit element



LDNF1B { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

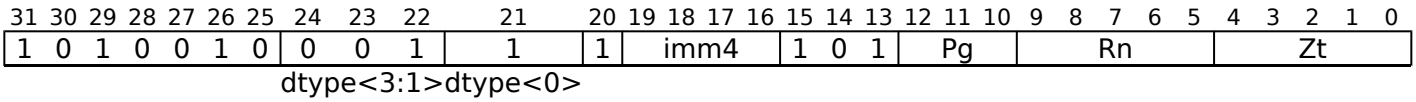
32-bit element



LDNF1B { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

64-bit element



LDNF1B { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

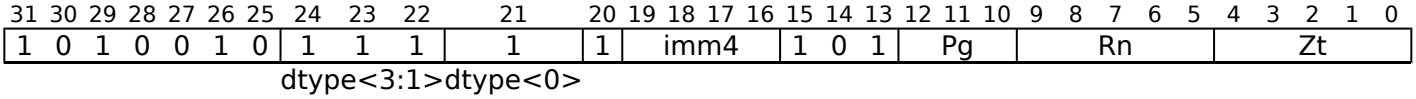
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDNF1D

Contiguous load non-fault doublewords to vector (immediate index)

Contiguous load with non-faulting behavior of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.



LDNF1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

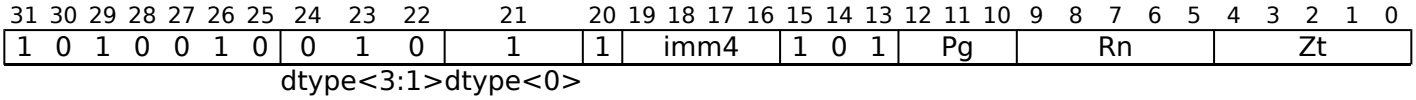
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDNF1H

Contiguous load non-fault unsigned halfwords to vector (immediate index)

Contiguous load with non-faulting behavior of unsigned halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 3 classes: 16-bit element , 32-bit element and 64-bit element

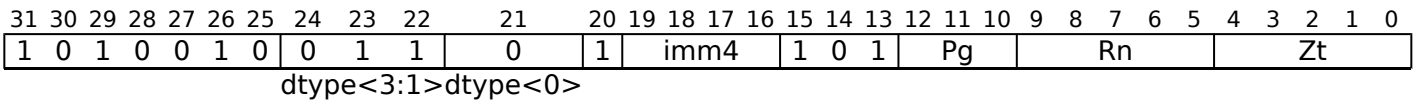
16-bit element



LDNF1H { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

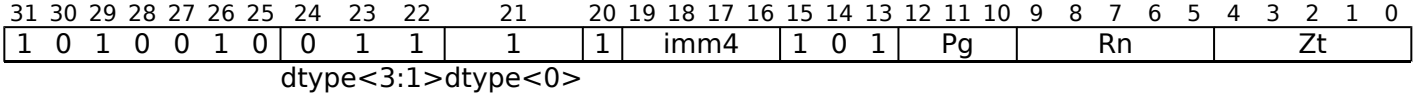
32-bit element



LDNF1H { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

64-bit element



LDNF1H { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

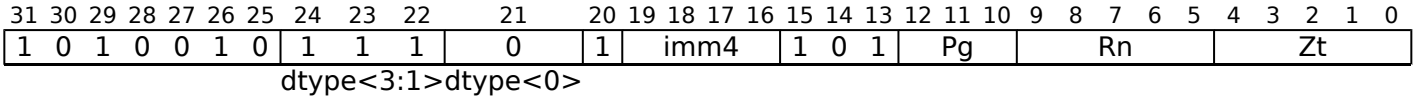
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDNF1SB

Contiguous load non-fault signed bytes to vector (immediate index)

Contiguous load with non-faulting behavior of signed bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 3 classes: 16-bit element , 32-bit element and 64-bit element

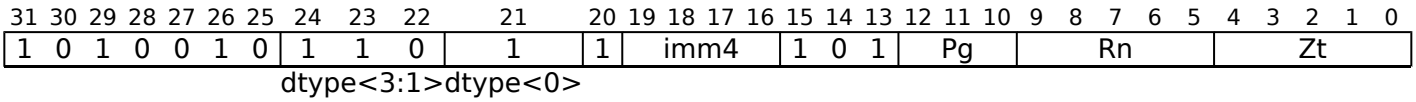
16-bit element



LDNF1SB { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
```

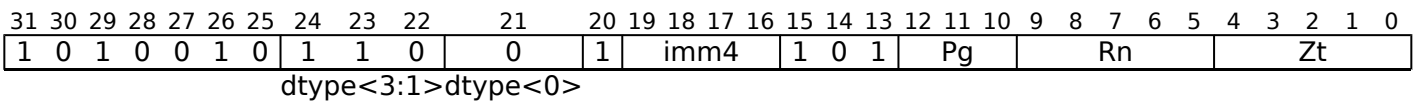
32-bit element



LDNF1SB { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
```

64-bit element



LDNF1SB { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

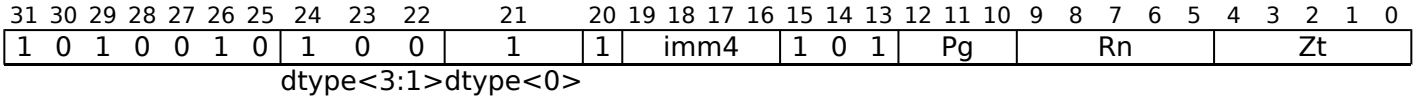
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDNF1SH

Contiguous load non-fault signed halfwords to vector (immediate index)

Contiguous load with non-faulting behavior of signed halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 2 classes: 32-bit element and 64-bit element

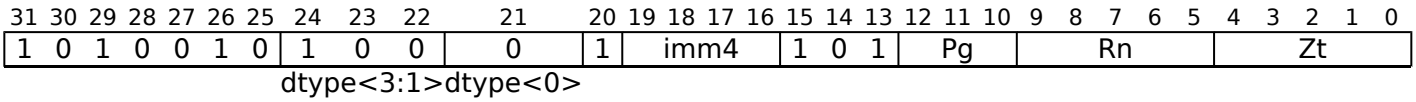
32-bit element



LDNF1SH { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
```

64-bit element



LDNF1SH { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

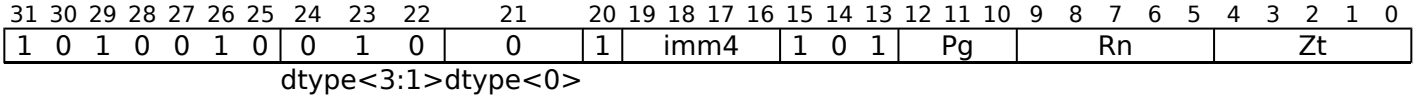
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDNF1SW

Contiguous load non-fault signed words to vector (immediate index)

Contiguous load with non-faulting behavior of signed words to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.



LDNF1SW { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = FALSE;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

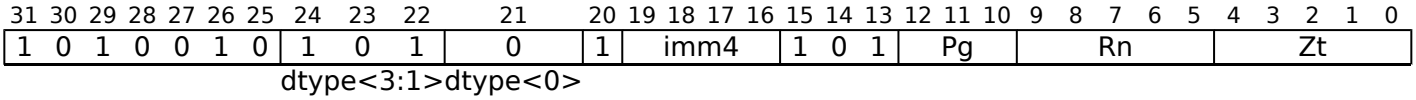
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDNF1W

Contiguous load non-fault unsigned words to vector (immediate index)

Contiguous load with non-faulting behavior of unsigned words to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 2 classes: 32-bit element and 64-bit element

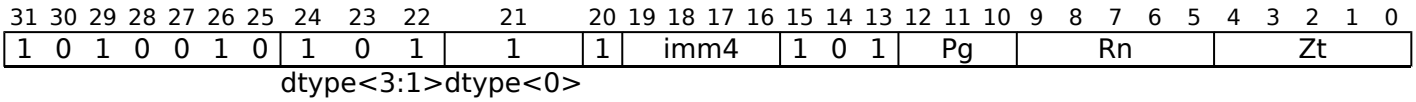
32-bit element



LDNF1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

64-bit element



LDNF1W { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = TRUE;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(VL) orig = Z[t, VL];
bits(msize) data;
constant integer mbytes = msize DIV 8;
boolean fault = FALSE;
boolean faulted = FALSE;
boolean unknown = FALSE;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        // MemNF[] will return fault=TRUE if access is not performed for any reason
        (data, fault) = MemNF[addr, mbytes, AccType_NONFAULT];
    else
        (data, fault) = (Zeros(msize), FALSE);

    // FFR elements set to FALSE following a supressed access/fault
    faulted = faulted || fault;
    if faulted then
        ElemFFR[e, esize] = '0';

    // Value becomes CONSTRAINED UNPREDICTABLE after an FFR element is FALSE
    unknown = unknown || ElemFFR[e, esize] == '0';
    if unknown then
        if !fault && ConstrainUnpredictableBool(Unpredictable_SVELDNFDATA) then
            Elem[result, e, esize] = Extend(data, esize, unsigned);
        elsif ConstrainUnpredictableBool(Unpredictable_SVELDNFZERO) then
            Elem[result, e, esize] = Zeros(esize);
        else  // merge
            Elem[result, e, esize] = Elem[orig, e, esize];
    else
        Elem[result, e, esize] = Extend(data, esize, unsigned);

Z[t, VL] = result;
```

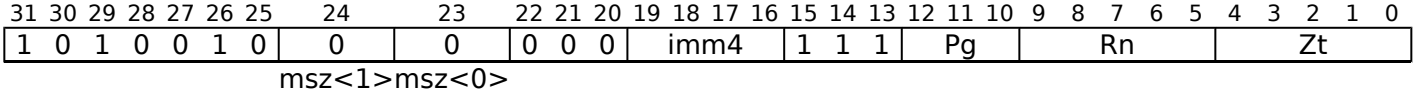
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDNT1B (scalar plus immediate)

Contiguous load non-temporal bytes to vector (immediate index)

Contiguous load non-temporal of bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.



LDNT1B { <Zt>.B }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

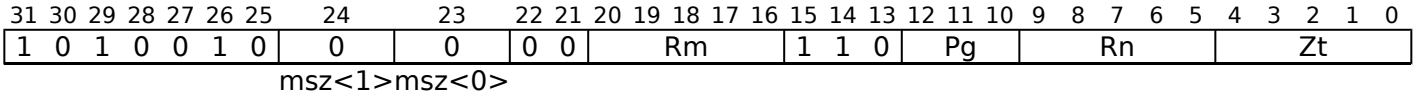
Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVESTREAM];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

LDNT1B (scalar plus scalar)

Contiguous load non-temporal bytes to vector (scalar index)

Contiguous load non-temporal of bytes to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.
A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.



LDNT1B { <Zt>.B }, <Pg>/Z, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

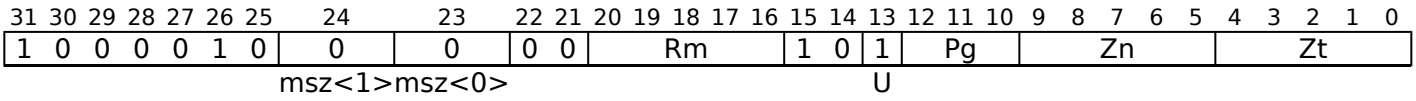
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(64) offset;
bits(PL) mask = P[g, PL];
bits(VL) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVESTREAM];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

LDNT1B (vector plus scalar)

Gather load non-temporal unsigned bytes

Gather load non-temporal of unsigned bytes to active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.
A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 2 classes: 32-bit unscaled offset and 64-bit unscaled offset

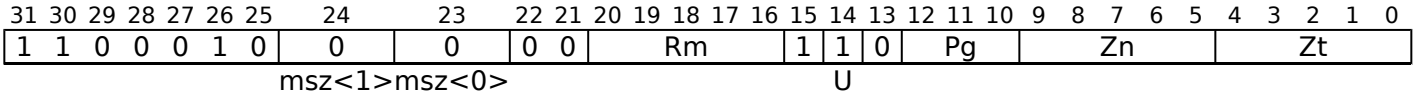
32-bit unscaled offset



LDNT1B { <Zt>.S }, <Pg>/Z, [<Zn>.S{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = TRUE;
```

64-bit unscaled offset



LDNT1B { <Zt>.D }, <Pg>/Z, [<Zn>.D{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the base scalable vector register, encoded in the "Zn" field.
<Xm>	Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
        data = Mem[addr, mbytes, AccType_SVESTREAM];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

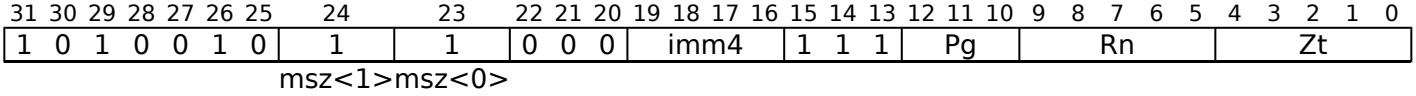
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDNT1D (scalar plus immediate)

Contiguous load non-temporal doublewords to vector (immediate index)

Contiguous load non-temporal of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.



LDNT1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

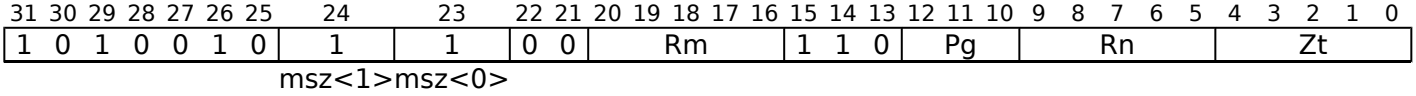
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVESTREAM];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

LDNT1D (scalar plus scalar)

Contiguous load non-temporal doublewords to vector (scalar index)

Contiguous load non-temporal of doublewords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.



LDNT1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(64) offset;
bits(PL) mask = P[g, PL];
bits(VL) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVESTREAM];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

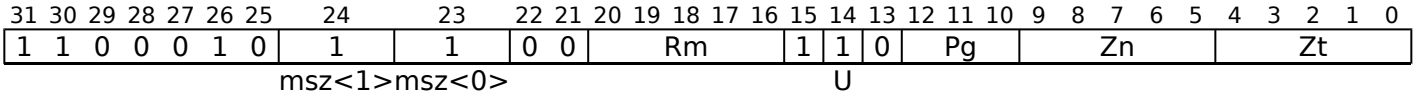
LDNT1D (vector plus scalar)

Gather load non-temporal unsigned doublewords

Gather load non-temporal of doublewords to active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.



LDNT1D { <Zt>.D }, <Pg>/Z, [<Zn>.D{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the base scalable vector register, encoded in the "Zn" field.
<Xm>	Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

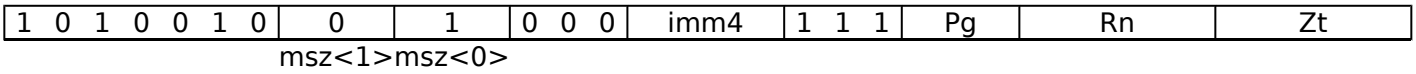
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
        data = Mem[addr, mbytes, AccType_SVESTREAM];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

LDNT1H (scalar plus immediate)

Contiguous load non-temporal halfwords to vector (immediate index)

Contiguous load non-temporal of halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.



LDNT1H { <Zt>.H }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

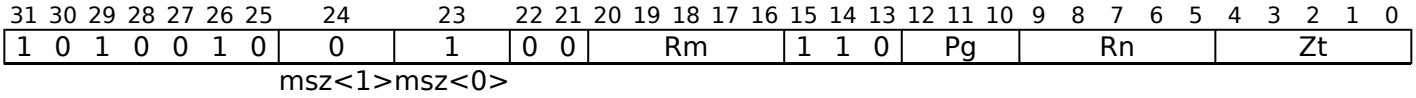
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVESTREAM];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

LDNT1H (scalar plus scalar)

Contiguous load non-temporal halfwords to vector (scalar index)

Contiguous load non-temporal of halfwords to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.



LDNT1H { <Zt>.H }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

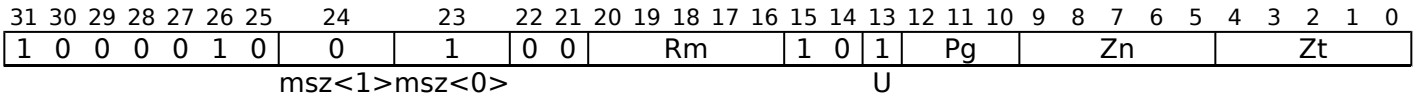
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(64) offset;
bits(PL) mask = P[g, PL];
bits(VL) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVESTREAM];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

LDNT1H (vector plus scalar)

Gather load non-temporal unsigned halfwords

Gather load non-temporal of unsigned halfwords to active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.
A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 2 classes: 32-bit unscaled offset and 64-bit unscaled offset

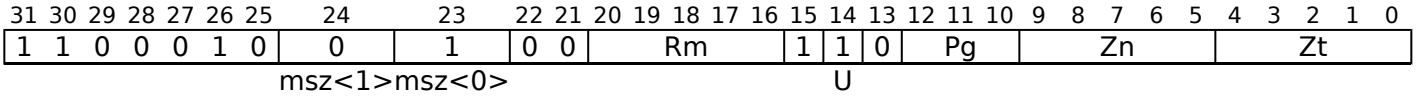
32-bit unscaled offset



LDNT1H { <Zt>.S }, <Pg>/Z, [<Zn>.S{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = TRUE;
```

64-bit unscaled offset



LDNT1H { <Zt>.D }, <Pg>/Z, [<Zn>.D{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the base scalable vector register, encoded in the "Zn" field.
<Xm>	Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
        data = Mem[addr, mbytes, AccType_SVESTREAM];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

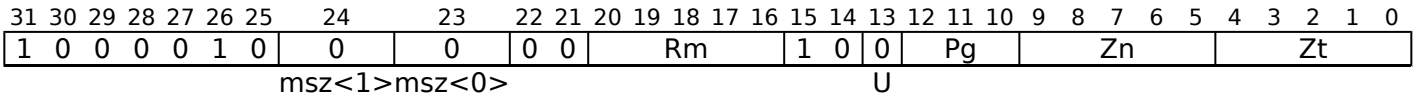
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDNT1SB

Gather load non-temporal signed bytes

Gather load non-temporal of signed bytes to active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.
A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 2 classes: 32-bit unscaled offset and 64-bit unscaled offset

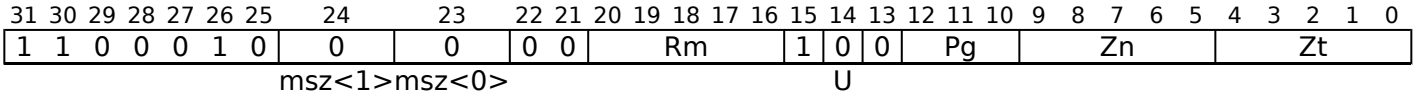
32-bit unscaled offset



LDNT1SB { <Zt>.S }, <Pg>/Z, [<Zn>.S{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
boolean unsigned = FALSE;
```

64-bit unscaled offset



LDNT1SB { <Zt>.D }, <Pg>/Z, [<Zn>.D{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the base scalable vector register, encoded in the "Zn" field.
<Xm>	Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
        data = Mem[addr, mbytes, AccType_SVESTREAM];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

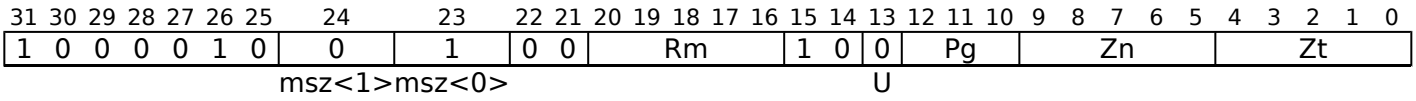
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDNT1SH

Gather load non-temporal signed halfwords

Gather load non-temporal of signed halfwords to active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.
A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 2 classes: 32-bit unscaled offset and 64-bit unscaled offset

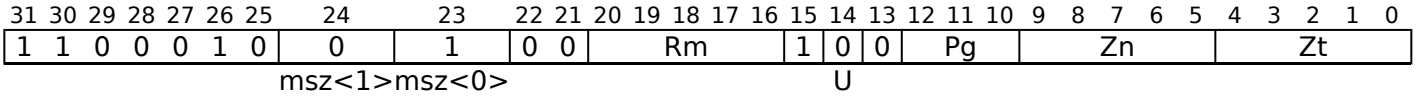
32-bit unscaled offset



LDNT1SH { <Zt>.S }, <Pg>/Z, [<Zn>.S{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
boolean unsigned = FALSE;
```

64-bit unscaled offset



LDNT1SH { <Zt>.D }, <Pg>/Z, [<Zn>.D{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the base scalable vector register, encoded in the "Zn" field.
<Xm>	Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

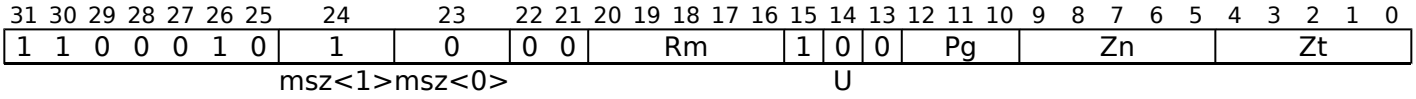
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
        data = Mem[addr, mbytes, AccType_SVESTREAM];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDNT1SW

Gather load non-temporal signed words

Gather load non-temporal of signed words to active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.
A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.



LDNT1SW { <Zt>.D }, <Pg>/Z, [<Zn>.D{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the base scalable vector register, encoded in the "Zn" field.
<Xm>	Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

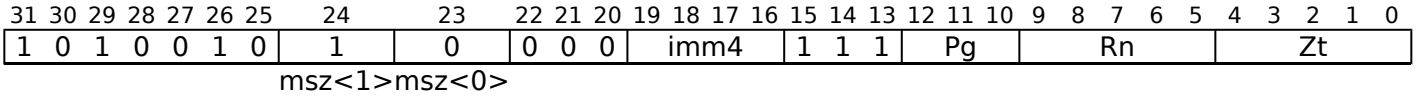
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
        data = Mem[addr, mbytes, AccType_SVESTREAM];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

LDNT1W (scalar plus immediate)

Contiguous load non-temporal words to vector (immediate index)

Contiguous load non-temporal of words to elements of a vector register from the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.



LDNT1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

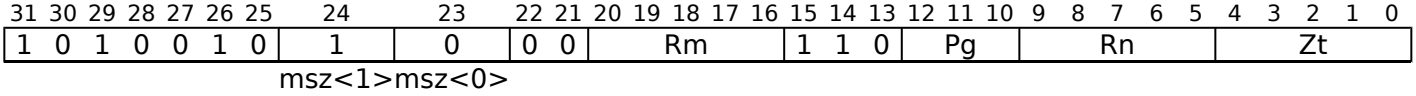
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVESTREAM];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

LDNT1W (scalar plus scalar)

Contiguous load non-temporal words to vector (scalar index)

Contiguous load non-temporal of words to elements of a vector register from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.



LDNT1W { <Zt>.S }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

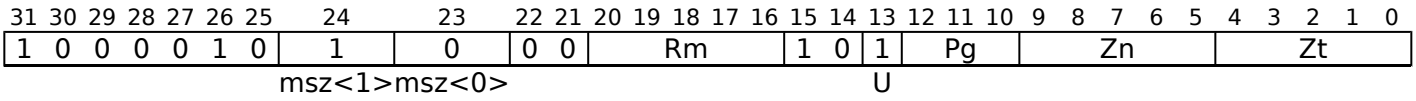
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(64) offset;
bits(PL) mask = P[g, PL];
bits(VL) result;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SVESTREAM];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

LDNT1W (vector plus scalar)

Gather load non-temporal unsigned words

Gather load non-temporal of unsigned words to active elements of a vector register from memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements will not cause a read from Device memory or signal faults, and are set to zero in the destination vector.
A non-temporal load is a hint to the system that this data is unlikely to be referenced again soon.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 2 classes: 32-bit unscaled offset and 64-bit unscaled offset

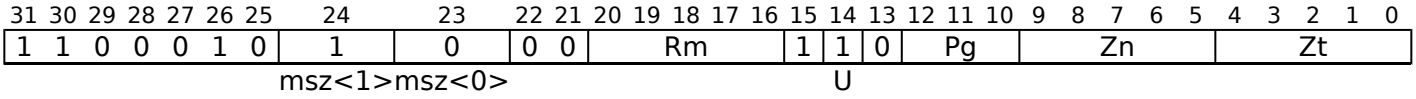
32-bit unscaled offset



LDNT1W { <Zt>.S }, <Pg>/Z, [<Zn>.S{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
boolean unsigned = TRUE;
```

64-bit unscaled offset



LDNT1W { <Zt>.D }, <Pg>/Z, [<Zn>.D{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the base scalable vector register, encoded in the "Zn" field.
<Xm>	Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) result;
bits(msize) data;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
        data = Mem[addr, mbytes, AccType_SVESTREAM];
        Elem[result, e, esize] = Extend(data, esize, unsigned);
    else
        Elem[result, e, esize] = Zeros(esize);

Z[t, VL] = result;
```

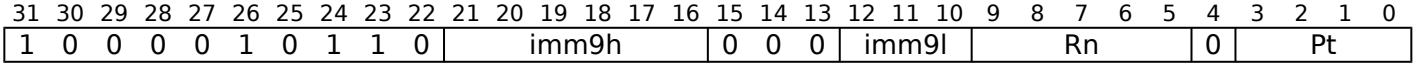
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDR (predicate)

Load predicate register

Load a predicate register from a memory address generated by a 64-bit scalar base, plus an immediate offset in the range -256 to 255 which is multiplied by the current predicate register size in bytes. This instruction is unpredicated.

The load is performed as contiguous byte accesses, each containing 8 consecutive predicate bits in ascending element order, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then a general-purpose base register must be aligned to 2 bytes.



LDR <Pt>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Pt);
integer n = UInt(Rn);
integer imm = SInt(imm9h:imm9l);
```

Assembler Symbols

<Pt>	Is the name of the destination scalable predicate register, encoded in the "Pt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9h:imm9l" fields.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = PL DIV 8;
bits(64) base;
integer offset = imm * elements;
bits(PL) result;

if n == 31 then
    CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n, 64];

boolean aligned = AArch64.CheckAlignment(base + offset, 2, AccType_SVE, FALSE);
for e = 0 to elements-1
    Elem[result, e, 8] = AArch64.MemSingle[base + offset, 1, AccType_SVE, aligned];
    offset = offset + 1;

P[t, PL] = result;
```

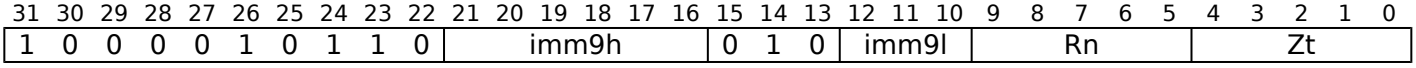
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDR (vector)

Load vector register

Load a vector register from a memory address generated by a 64-bit scalar base, plus an immediate offset in the range -256 to 255 which is multiplied by the current vector register size in bytes. This instruction is unpredicated.

The load is performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then the base register must be aligned to 16 bytes.



LDR <Zt>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer imm = SInt(imm9h:imm9l);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9h:imm9l" fields.

Operation

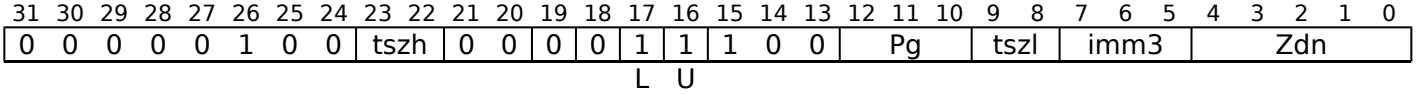
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 8;
bits(64) base;
integer offset = imm * elements;
bits(VL) result;

if n == 31 then
    CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n, 64];

boolean aligned = AArch64.CheckAlignment(base + offset, 16, AccType_SVE, FALSE);
for e = 0 to elements-1
    Elem[result, e, 8] = AArch64.MemSingle[base + offset, 1, AccType_SVE, aligned];
    offset = offset + 1;

Z[t, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LSL (immediate, predicated)

Logical shift left by immediate (predicated)

Shift left by immediate each active element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. Inactive elements in the destination vector register remain unmodified.



LSL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
case tsize of
    when '0000' UNDEFINED;
    when '0001' esize = 8;
    when '001x' esize = 16;
    when '01xx' esize = 32;
    when '1xxx' esize = 64;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = UInt(tsize:imm3) - esize;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer PL = VL DIV 8;
bits(VL) operand1 = Z[dn, VL];
bits(PL) mask = P[g, PL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = LSL(element1, shift);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

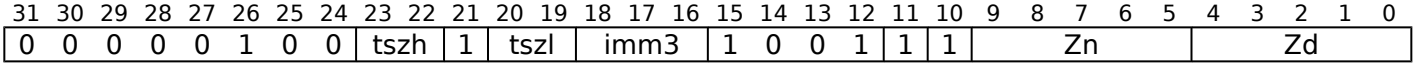
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LSL (immediate, unpredicated)

Logical shift left by immediate (unpredicated)

Shift left by immediate each element of the source vector, and place the results in the corresponding elements of the destination vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. This instruction is unpredicated.



LSL <Zd>.<T>, <Zn>.<T>, #<const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
case tsize of
    when '0000' UNDEFINED;
    when '0001' esize = 8;
    when '001x' esize = 16;
    when '01xx' esize = 32;
    when '1xxx' esize = 64;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = UInt(tsize:imm3) - esize;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    Elem[result, e, esize] = LSL(element1, shift);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

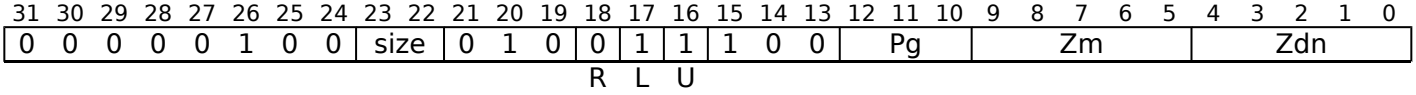
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LSL (vectors)

Logical shift left by vector (predicated)

Shift left active elements of the first source vector by corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.



LSL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        integer shift = Min(UInt(element2), esize);
        Elem[result, e, esize] = LSL(element1, shift);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

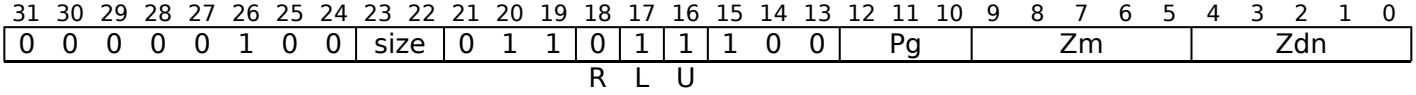
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LSL (wide elements, predicated)

Logical shift left by 64-bit wide elements (predicated)

Shift left active elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. Inactive elements in the destination vector register remain unmodified.



LSL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	RESERVED

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(64) element2 = Elem[operand2, (e * esize) DIV 64, 64];
        integer shift = Min(UInt(element2), esize);
        Elem[result, e, esize] = LSL(element1, shift);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

        - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
        - The values of the NZCV flags.
    - The response of this instruction to asynchronous exceptions does not vary based on:
        - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
        - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

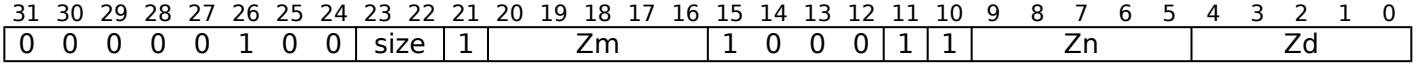
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and destination element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LSL (wide elements, unpredicated)

Logical shift left by 64-bit wide elements (unpredicated)

Shift left all elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and place the first in the corresponding elements of the destination vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. Inactive elements in the destination vector register remain unmodified.



LSL <Zd>.<T>, <Zn>.<T>, <Zm>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	RESERVED

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(64) element2 = Elem[operand2, (e * esize) DIV 64, 64];
    integer shift = Min(UInt(element2), esize);
    Elem[result, e, esize] = LSL(element1, shift);

Z[d, VL] = result;
```

Operational information

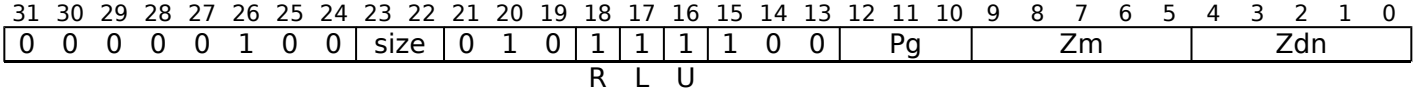
If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

LSLR

Reversed logical shift left by vector (predicated)

Reversed shift left active elements of the second source vector by corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.



LSLR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        integer shift = Min(UInt(element1), esize);
        Elem[result, e, esize] = LSL(element2, shift);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

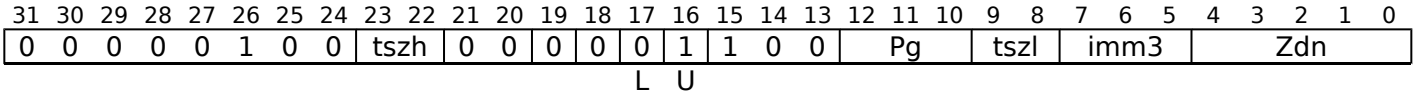
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LSR (immediate, predicated)

Logical shift right by immediate (predicated)

Shift right by immediate, inserting zeroes, each active element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. Inactive elements in the destination vector register remain unmodified.



LSR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
case tsize of
    when '0000' UNDEFINED;
    when '0001' esize = 8;
    when '001x' esize = 16;
    when '01xx' esize = 32;
    when '1xxx' esize = 64;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer PL = VL DIV 8;
bits(VL) operand1 = Z[dn, VL];
bits(PL) mask = P[g, PL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = LSR(element1, shift);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

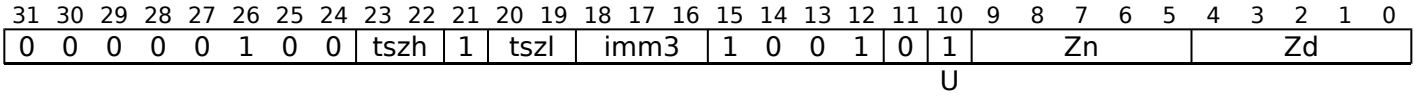
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LSR (immediate, unpredicated)

Logical shift right by immediate (unpredicated)

Shift right by immediate, inserting zeroes, each element of the source vector, and place the results in the corresponding elements of the destination vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



LSR <Zd>.<T>, <Zn>.<T>, #<const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
case tsize of
    when '0000' UNDEFINED;
    when '0001' esize = 8;
    when '001x' esize = 16;
    when '01xx' esize = 32;
    when '1xxx' esize = 64;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    Elem[result, e, esize] = LSR(element1, shift);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

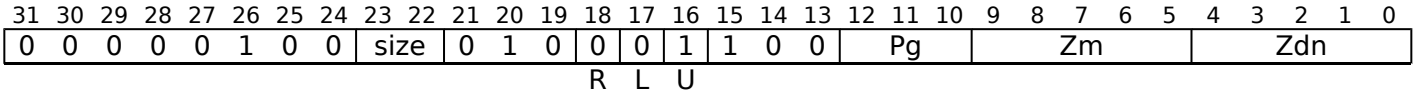
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LSR (vectors)

Logical shift right by vector (predicated)

Shift right, inserting zeroes, active elements of the first source vector by corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.



LSR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        integer shift = Min(UInt(element2), esize);
        Elem[result, e, esize] = LSR(element1, shift);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

        - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
        - The values of the NZCV flags.
    - The response of this instruction to asynchronous exceptions does not vary based on:
        - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
        - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

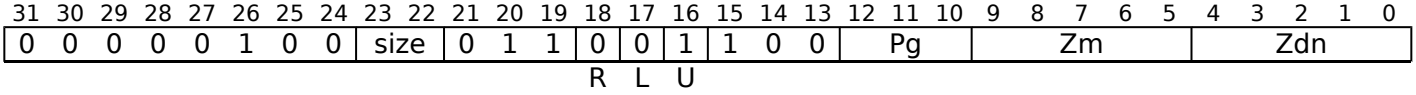
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LSR (wide elements, predicated)

Logical shift right by 64-bit wide elements (predicated)

Shift right, inserting zeroes, active elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. Inactive elements in the destination vector register remain unmodified.



LSR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	RESERVED

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(64) element2 = Elem[operand2, (e * esize) DIV 64, 64];
        integer shift = Min(UInt(element2), esize);
        Elem[result, e, esize] = LSR(element1, shift);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

        - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
        - The values of the NZCV flags.
    - The response of this instruction to asynchronous exceptions does not vary based on:
        - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
        - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

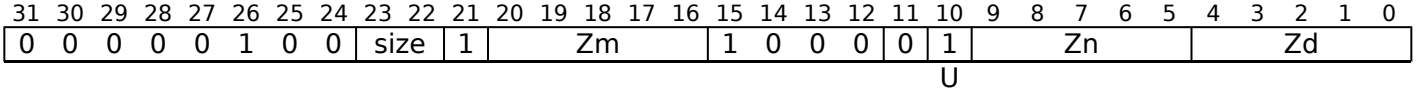
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and destination element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LSR (wide elements, unpredicated)

Logical shift right by 64-bit wide elements (unpredicated)

Shift right, inserting zeroes, all elements of the first source vector by corresponding overlapping 64-bit elements of the second source vector and place the first in the corresponding elements of the destination vector. The shift amount is a vector of unsigned 64-bit doubleword elements in which all bits are significant, and not used modulo the destination element size. This instruction is unpredicated.



LSR <Zd>.<T>, <Zn>.<T>, <Zm>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	RESERVED

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(64) element2 = Elem[operand2, (e * esize) DIV 64, 64];
    integer shift = Min(UInt(element2), esize);
    Elem[result, e, esize] = LSR(element1, shift);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

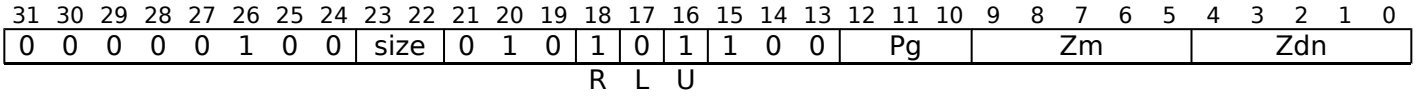
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.

- The values of the NZCV flags.

LSRR

Reversed logical shift right by vector (predicated)

Reversed shift right, inserting zeroes, active elements of the second source vector by corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. The shift amount operand is a vector of unsigned elements in which all bits are significant, and not used modulo the element size. Inactive elements in the destination vector register remain unmodified.



LSRR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element1 = Elem[operand1, e, esize];
        bits(esize) element2 = Elem[operand2, e, esize];
        integer shift = Min(UInt(element1), esize);
        Elem[result, e, esize] = LSR(element2, shift);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

        - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
        - The values of the NZCV flags.
    - The response of this instruction to asynchronous exceptions does not vary based on:
        - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
        - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

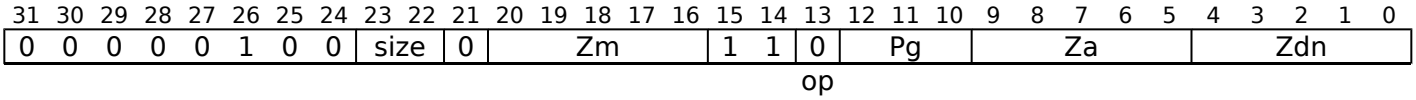
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MAD

Multiply-add vectors (predicated), writing multiplicand [Zdn = Za + Zdn * Zm]

Multiply the corresponding active elements of the first and second source vectors and add to elements of the third (addend) vector. Destructively place the results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.



MAD <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Za>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
integer a = UInt(Za);
boolean sub_op = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Za> Is the name of the third source scalable vector register, encoded in the "Za" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = if AnyActiveElement(mask, esize) then Z[a, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element1 = UInt(Elem[operand1, e, esize]);
        integer element2 = UInt(Elem[operand2, e, esize]);
        integer product = element1 * element2;
        if sub_op then
            Elem[result, e, esize] = Elem[operand3, e, esize] - product;
        else
            Elem[result, e, esize] = Elem[operand3, e, esize] + product;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

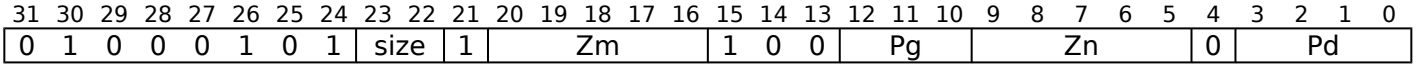
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MATCH

Detect any matching elements, setting the condition flags

This instruction compares each active 8-bit or 16-bit character in the first source vector with all of the characters in the corresponding 128-bit segment of the second source vector. Where the first source element detects any matching characters in the second segment it places true in the corresponding element of the destination predicate, otherwise false. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.



MATCH <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE2() then UNDEFINED;
if size IN {'1x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer d = UInt(Pd);
integer n = UInt(Zn);
integer m = UInt(Zm);
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size<0>”:

size<0>	<T>
0	B
1	H

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(PL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer segmentbase = e - (e MOD eltspersegment);
        ElemP[result, e, esize] = '0';
        bits(esize) element1 = Elem[operand1, e, esize];
        for i = segmentbase to (segmentbase + eltspersegment) - 1
            bits(esize) element2 = Elem[operand2, i, esize];
            if element1 == element2 then
                ElemP[result, e, esize] = '1';
    else
        ElemP[result, e, esize] = '0';

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

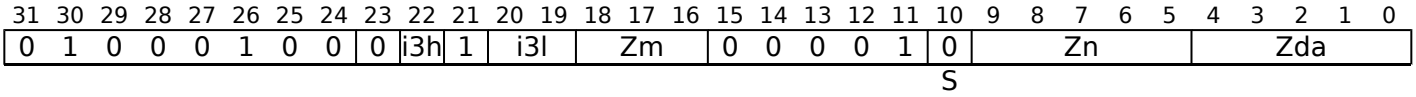
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MLA (indexed)

Multiply-add to accumulator (indexed)

Multiply all integer elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The products are then destructively added to the corresponding elements of the addend and destination vector.
The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element. This instruction is unpredicated.
It has encodings from 3 classes: 16-bit , 32-bit and 64-bit

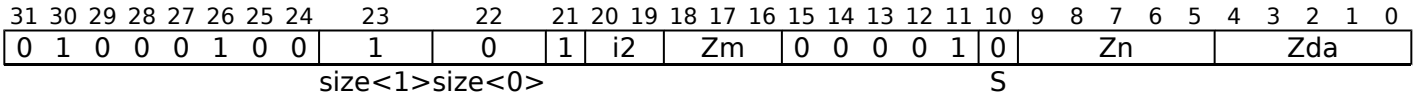
16-bit



MLA <Zda>.H, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

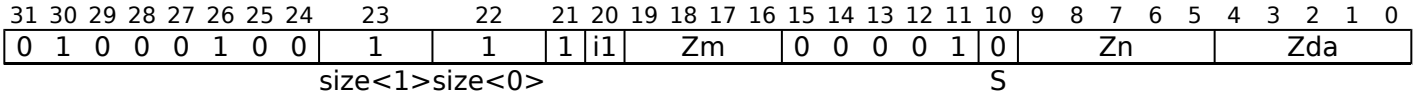
32-bit



MLA <Zda>.S, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

64-bit



MLA <Zda>.D, <Zn>.D, <Zm>.D[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the 16-bit and 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the 16-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.

For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, s, esize]);
    bits(esize) product = (element1 * element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + product;

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

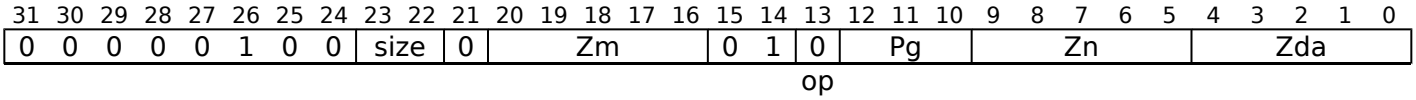
- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MLA (vectors)

Multiply-add vectors (predicated), writing addend [Zda = Zda + Zn * Zm]

Multiply the corresponding active elements of the first and second source vectors and add to elements of the third source (addend) vector. Destructively place the results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.



MLA <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean sub_op = FALSE;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element1 = UInt(Elem[operand1, e, esize]);
        integer element2 = UInt(Elem[operand2, e, esize]);
        integer product = element1 * element2;
        if sub_op then
            Elem[result, e, esize] = Elem[operand3, e, esize] - product;
        else
            Elem[result, e, esize] = Elem[operand3, e, esize] + product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize];

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

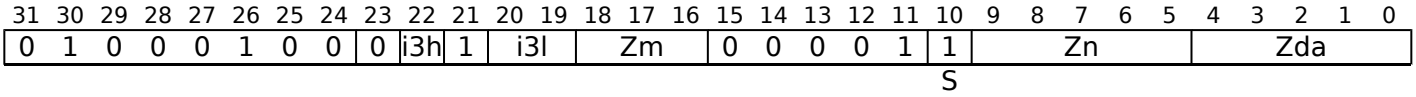
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MLS (indexed)

Multiply-subtract from accumulator (indexed)

Multiply all integer elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The products are then destructively subtracted from the corresponding elements of the addend and destination vector.
The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element. This instruction is unpredicated.
It has encodings from 3 classes: 16-bit , 32-bit and 64-bit

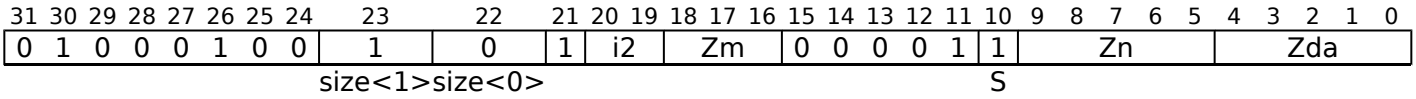
16-bit



MLS <Zda>.H, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

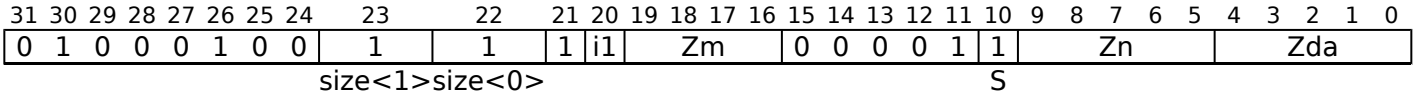
32-bit



MLS <Zda>.S, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

64-bit



MLS <Zda>.D, <Zn>.D, <Zm>.D[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the 16-bit and 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the 16-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.

For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, s, esize]);
    bits(esize) product = (element1 * element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] - product;

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

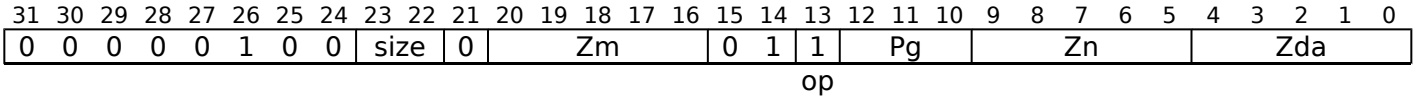
- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MLS (vectors)

Multiply-subtract vectors (predicated), writing addend [Zda = Zda - Zn * Zm]

Multiply the corresponding active elements of the first and second source vectors and subtract from elements of the third source (addend) vector. Destructively place the results in the destination and third source (addend) vector. Inactive elements in the destination vector register remain unmodified.



MLS <Zda>.<T>, <Pg>/M, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean sub_op = TRUE;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element1 = UInt(Elem[operand1, e, esize]);
        integer element2 = UInt(Elem[operand2, e, esize]);
        integer product = element1 * element2;
        if sub_op then
            Elem[result, e, esize] = Elem[operand3, e, esize] - product;
        else
            Elem[result, e, esize] = Elem[operand3, e, esize] + product;
    else
        Elem[result, e, esize] = Elem[operand3, e, esize];

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

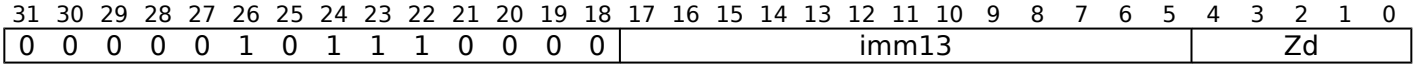
MOV

Move logical bitmask immediate to vector (unpredicated)

Unconditionally broadcast the logical bitmask immediate into each element of the destination vector. This instruction is unpredicated. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits.

This is an alias of DUPM. This means:

- The encodings in this description are named to match the encodings of DUPM.
- The description of DUPM gives the operational pseudocode for this instruction.



MOV <Zd>.<T>, #<const>

is equivalent to

DUPM <Zd>.<T>, #<const>

and is the preferred disassembly when SVEMoveMaskPreferred(imm13).

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “imm13<12>:imm13<5:0>”:

imm13<12>	imm13<5:0>	<T>
0	0xxxxx	S
0	10xxxx	H
0	110xxx	B
0	1110xx	B
0	11110x	B
0	111110	RESERVED
0	111111	RESERVED
1	xxxxxx	D

<const> Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the "imm13" field.

Operation

The description of DUPM gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

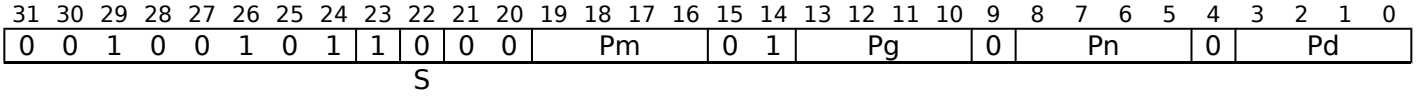
MOV

Move predicate (unpredicated)

Read all elements from the source predicate and place in the destination predicate. This instruction is unpredicated. Does not set the condition flags.

This is an alias of ORR (predicates). This means:

- The encodings in this description are named to match the encodings of ORR (predicates).
- The description of ORR (predicates) gives the operational pseudocode for this instruction.



MOV <Pd>.B, <Pn>.B

is equivalent to

ORR <Pd>.B, <Pn>/Z, <Pn>.B, <Pn>.B

and is the preferred disassembly when `S == '0' && Pn == Pm && Pm == Pg`.

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

The description of ORR (predicates) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

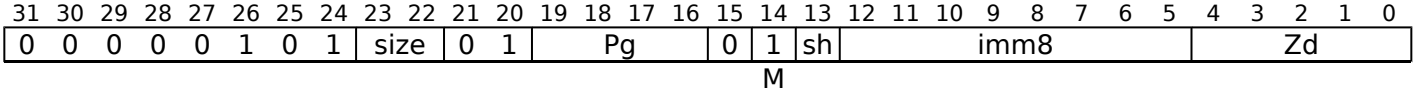
MOV (immediate, predicated, merging)

Move signed integer immediate to vector elements (merging)

Move a signed integer immediate to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.
The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).
The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<simm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

This is an alias of CPY (immediate, merging). This means:

- The encodings in this description are named to match the encodings of CPY (immediate, merging).
- The description of CPY (immediate, merging) gives the operational pseudocode for this instruction.



MOV <Zd>.<T>, <Pg>/M, #<imm>{, <shift>}

is equivalent to

CPY <Zd>.<T>, <Pg>/M, #<imm>{, <shift>}

and is always the preferred disassembly.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<imm> Is a signed immediate in the range -128 to 127, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #8

Operation

The description of CPY (immediate, merging) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

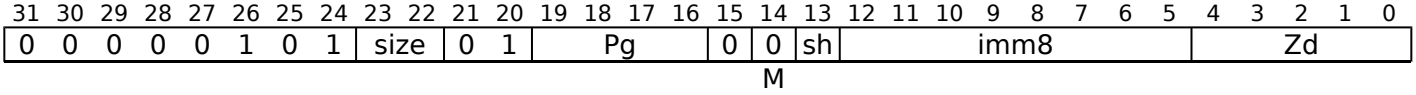
MOV (immediate, predicated, zeroing)

Move signed integer immediate to vector elements (zeroing)

Move a signed integer immediate to each active element in the destination vector. Inactive elements in the destination vector register are set to zero.
The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).
The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<simm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

This is an alias of CPY (immediate, zeroing). This means:

- The encodings in this description are named to match the encodings of CPY (immediate, zeroing).
- The description of CPY (immediate, zeroing) gives the operational pseudocode for this instruction.



MOV <Zd>.<T>, <Pg>/Z, #<imm>{, <shift>}

is equivalent to

CPY <Zd>.<T>, <Pg>/Z, #<imm>{, <shift>}

and is always the preferred disassembly.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register, encoded in the "Pg" field.

<imm> Is a signed immediate in the range -128 to 127, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #8

Operation

The description of CPY (immediate, zeroing) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

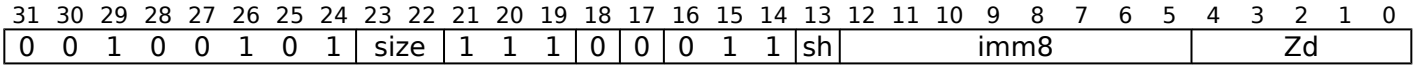
MOV (immediate, unpredicated)

Move signed immediate to vector elements (unpredicated)

Unconditionally broadcast the signed integer immediate into each element of the destination vector. This instruction is unpredicated.
The immediate operand is a signed value in the range -128 to +127, and for element widths of 16 bits or higher it may also be a signed multiple of 256 in the range -32768 to +32512 (excluding 0).
The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<simm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".

This is an alias of DUP (immediate). This means:

- The encodings in this description are named to match the encodings of DUP (immediate).
- The description of DUP (immediate) gives the operational pseudocode for this instruction.



MOV <Zd>.<T>, #<imm>{, <shift>}

is equivalent to

DUP <Zd>.<T>, #<imm>{, <shift>}

and is always the preferred disassembly.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<imm> Is a signed immediate in the range -128 to 127, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #8

Operation

The description of DUP (immediate) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

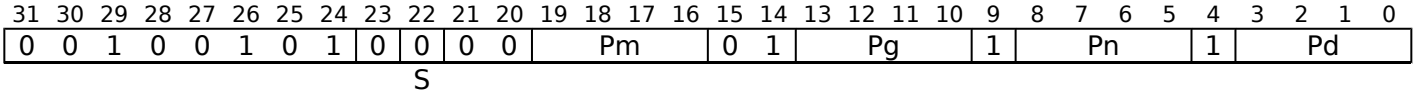
MOV (predicate, predicated, merging)

Move predicates (merging)

Read active elements from the source predicate and place in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register remain unmodified. Does not set the condition flags.

This is an alias of SEL (predicates). This means:

- The encodings in this description are named to match the encodings of SEL (predicates).
- The description of SEL (predicates) gives the operational pseudocode for this instruction.



MOV <Pd>.B, <Pg>/M, <Pn>.B

is equivalent to

SEL <Pd>.B, <Pg>, <Pn>.B, <Pd>.B

and is the preferred disassembly when `Pd == Pm`.

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

The description of SEL (predicates) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

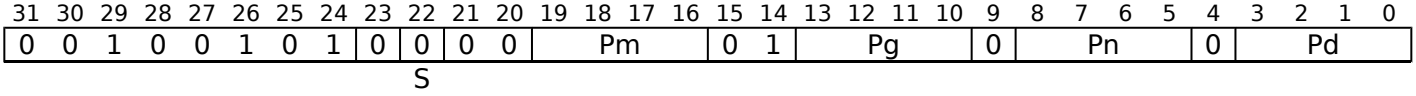
MOV (predicate, predicated, zeroing)

Move predicates (zeroing)

Read active elements from the source predicate and place in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is an alias of AND (predicates). This means:

- The encodings in this description are named to match the encodings of AND (predicates).
- The description of AND (predicates) gives the operational pseudocode for this instruction.



MOV <Pd>.B, <Pg>/Z, <Pn>.B

is equivalent to

AND <Pd>.B, <Pg>/Z, <Pn>.B, <Pn>.B

and is the preferred disassembly when `S == '0' && Pn == Pm`.

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

The description of AND (predicates) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

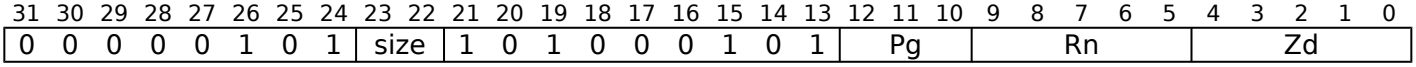
MOV (scalar, predicated)

Move general-purpose register to vector elements (predicated)

Move the general-purpose scalar source register to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This is an alias of CPY (scalar). This means:

- The encodings in this description are named to match the encodings of CPY (scalar).
- The description of CPY (scalar) gives the operational pseudocode for this instruction.



MOV <Zd>.<T>, <Pg>/M, <R><n|SP>

is equivalent to

CPY <Zd>.<T>, <Pg>/M, <R><n|SP>

and is always the preferred disassembly.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<R> Is a width specifier, encoded in “size”:

size	<R>
01	W
x0	W
11	X

<n|SP> Is the number [0-30] of the general-purpose source register or the name SP (31), encoded in the "Rn" field.

Operation

The description of CPY (scalar) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

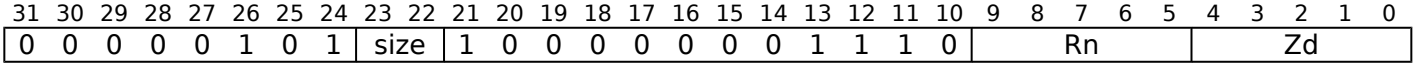
MOV (scalar, unpredicated)

Move general-purpose register to vector elements (unpredicated)

Unconditionally broadcast the general-purpose scalar source register into each element of the destination vector. This instruction is unpredicated.

This is an alias of DUP (scalar). This means:

- The encodings in this description are named to match the encodings of DUP (scalar).
- The description of DUP (scalar) gives the operational pseudocode for this instruction.



MOV <Zd>.<T>, <R><n|SP>

is equivalent to

DUP <Zd>.<T>, <R><n|SP>

and is always the preferred disassembly.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<R> Is a width specifier, encoded in “size”:

size	<R>
01	W
x0	W
11	X

<n|SP> Is the number [0-30] of the general-purpose source register or the name SP (31), encoded in the "Rn" field.

Operation

The description of DUP (scalar) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

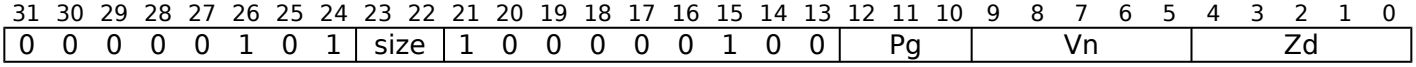
MOV (SIMD&FP scalar, predicated)

Move SIMD&FP scalar register to vector elements (predicated)

Move the SIMD & floating-point scalar source register to each active element in the destination vector. Inactive elements in the destination vector register remain unmodified.

This is an alias of CPY (SIMD&FP scalar). This means:

- The encodings in this description are named to match the encodings of CPY (SIMD&FP scalar).
- The description of CPY (SIMD&FP scalar) gives the operational pseudocode for this instruction.



MOV <Zd>.<T>, <Pg>/M, <V><n>

is equivalent to

CPY <Zd>.<T>, <Pg>/M, <V><n>

and is always the preferred disassembly.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<n> Is the number [0-31] of the source SIMD&FP register, encoded in the "Vn" field.

Operation

The description of CPY (SIMD&FP scalar) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

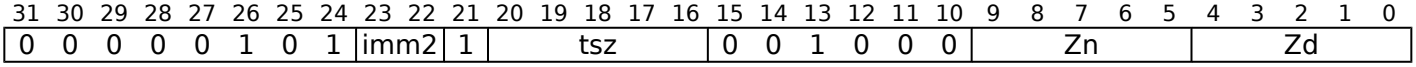
MOV (SIMD&FP scalar, unpredicated)

Move indexed element or SIMD&FP scalar to vector (unpredicated)

Unconditionally broadcast the SIMD&FP scalar into each element of the destination vector. This instruction is unpredicated.

This is an alias of DUP (indexed). This means:

- The encodings in this description are named to match the encodings of DUP (indexed).
- The description of DUP (indexed) gives the operational pseudocode for this instruction.



MOV <Zd>.<T>, <Zn>.<T>[<imm>]

is equivalent to

DUP <Zd>.<T>, <Zn>.<T>[<imm>]

and is the preferred disassembly when `BitCount(imm2:tsz) > 1`.

MOV <Zd>.<T>, <V><n>

is equivalent to

DUP <Zd>.<T>, <Zn>.<T>[0]

and is the preferred disassembly when `BitCount(imm2:tsz) == 1`.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tsz”:

tsz	<T>
00000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D
10000	Q

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<imm> Is the immediate index, in the range 0 to one less than the number of elements in 512 bits, encoded in "imm2:tsz".

<V> Is a width specifier, encoded in “tsz”:

tsz	<V>
00000	RESERVED
xxxx1	B
xxx10	H
xx100	S
x1000	D
10000	Q

<n> Is the number [0-31] of the source SIMD&FP register, encoded in the "Zn" field.

Operation

The description of DUP (indexed) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

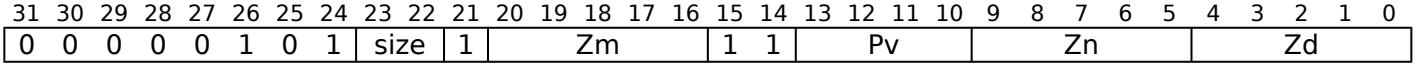
MOV (vector, predicated)

Move vector elements (predicated)

Move elements from the source vector to the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

This is an alias of SEL (vectors). This means:

- The encodings in this description are named to match the encodings of SEL (vectors).
- The description of SEL (vectors) gives the operational pseudocode for this instruction.



MOV <Zd>.<T>, <Pv>/M, <Zn>.<T>

is equivalent to

SEL <Zd>.<T>, <Pv>, <Zn>.<T>, <Zd>.<T>

and is the preferred disassembly when `Zd == Zm`.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pv> Is the name of the vector select predicate register, encoded in the "Pv" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

Operation

The description of SEL (vectors) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

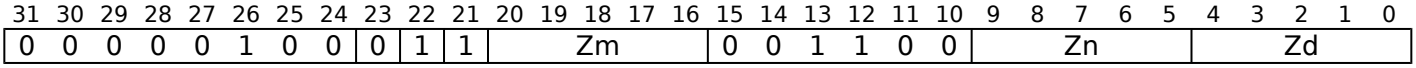
MOV (vector, unpredicated)

Move vector register (unpredicated)

Move vector register. This instruction is unpredicated.

This is an alias of ORR (vectors, unpredicated). This means:

- The encodings in this description are named to match the encodings of ORR (vectors, unpredicated).
- The description of ORR (vectors, unpredicated) gives the operational pseudocode for this instruction.



MOV <Zd>.D, <Zn>.D

is equivalent to

ORR <Zd>.D, <Zn>.D, <Zn>.D

and is the preferred disassembly when `Zn == Zm`.

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.

Operation

The description of ORR (vectors, unpredicated) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

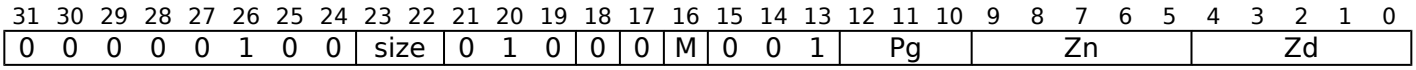
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MOVPRFX (predicated)

Move prefix (predicated)

The predicated MOVPRFX instruction is a hint to hardware that the instruction may be combined with the destructive instruction which follows it in program order to create a single constructive operation. Since it is a hint it is also permitted to be implemented as a discrete vector copy, and the result of executing the pair of instructions with or without combining is identical. The choice of combined versus discrete operation may vary dynamically.
Unless the combination of a constructive operation with merging predication is specifically required, it is strongly recommended that for performance reasons software should prefer to use the zeroing form of predicated MOVPRFX or the unpredicated MOVPRFX instruction.
Although the operation of the instruction is defined as a simple predicated vector copy, it is required that the prefixed instruction at PC+4 must be an SVE destructive binary or ternary instruction encoding, or a unary operation with merging predication, but excluding other MOVPRFX instructions. The prefixed instruction must specify the same predicate register, and have the same maximum element size (ignoring a fixed 64-bit "wide vector" operand), and the same destination vector as the MOVPRFX instruction. The prefixed instruction must not use the destination register in any other operand position, even if they have different names but refer to the same architectural register state. Any other use is UNPREDICTABLE.



MOVPRFX <Zd>.<T>, <Pg>/<ZM>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean merging = (M == '1');
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<ZM> Is the predication qualifier, encoded in “M”:

M	<ZM>
0	Z
1	M

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) dest = Z[d, VL];
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand1, e, esize];
        Elem[result, e, esize] = element;
    elsif merging then
        Elem[result, e, esize] = Elem[dest, e, esize];
    else
        Elem[result, e, esize] = Zeros(esize);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

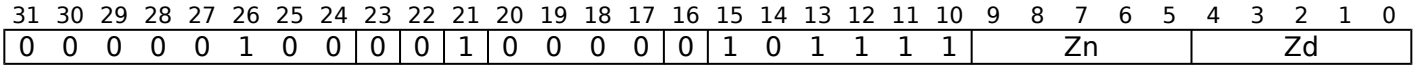
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MOVPRFX (unpredicated)

Move prefix (unpredicated)

The unpredicated MOVPRFX instruction is a hint to hardware that the instruction may be combined with the destructive instruction which follows it in program order to create a single constructive operation. Since it is a hint it is also permitted to be implemented as a discrete vector copy, and the result of executing the pair of instructions with or without combining is identical. The choice of combined versus discrete operation may vary dynamically.

Although the operation of the instruction is defined as a simple unpredicated vector copy, it is required that the prefixed instruction at PC+4 must be an SVE destructive binary or ternary instruction encoding, or a unary operation with merging predication, but excluding other MOVPRFX instructions. The prefixed instruction must specify the same destination vector as the MOVPRFX instruction. The prefixed instruction must not use the destination register in any other operand position, even if they have different names but refer to the same architectural register state. Any other use is UNPREDICTABLE.



MOVPRFX <Zd>, <Zn>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn>	Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(VL) result = Z[n, VL];
Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

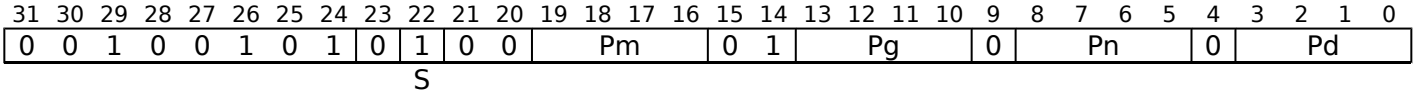
MOVS (predicated)

Move predicates (zeroing), setting the condition flags

Read active elements from the source predicate and place in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is an alias of ANDS. This means:

- The encodings in this description are named to match the encodings of ANDS.
- The description of ANDS gives the operational pseudocode for this instruction.



MOVS <Pd>.B, <Pg>/Z, <Pn>.B

is equivalent to

ANDS <Pd>.B, <Pg>/Z, <Pn>.B, <Pn>.B

and is the preferred disassembly when `S == '1' && Pn == Pm`.

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

The description of ANDS gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

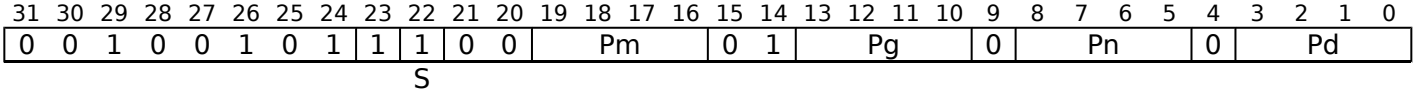
MOVS (unpredicated)

Move predicate (unpredicated), setting the condition flags

Read all elements from the source predicate and place in the destination predicate. This instruction is unpredicated. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is an alias of ORRS. This means:

- The encodings in this description are named to match the encodings of ORRS.
- The description of ORRS gives the operational pseudocode for this instruction.



MOVS <Pd>.B, <Pn>.B

is equivalent to

ORRS <Pd>.B, <Pn>/Z, <Pn>.B, <Pn>.B

and is the preferred disassembly when `S == '1' && Pn == Pm && Pm == Pg`.

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

The description of ORRS gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

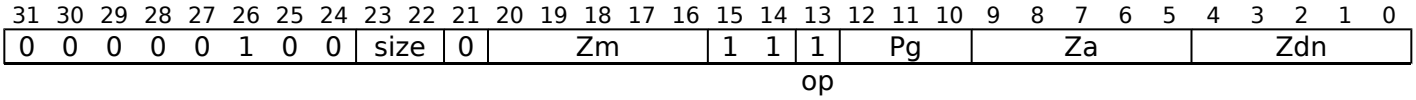
If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MSB

Multiply-subtract vectors (predicated), writing multiplicand [Zdn = Za - Zdn * Zm]

Multiply the corresponding active elements of the first and second source vectors and subtract from elements of the third (addend) vector. Destructively place the results in the destination and first source (multiplicand) vector. Inactive elements in the destination vector register remain unmodified.



MSB <Zdn>.<T>, <Pg>/M, <Zm>.<T>, <Za>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
integer a = UInt(Za);
boolean sub_op = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Za> Is the name of the third source scalable vector register, encoded in the "Za" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand3 = if AnyActiveElement(mask, esize) then Z[a, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element1 = UInt(Elem[operand1, e, esize]);
        integer element2 = UInt(Elem[operand2, e, esize]);
        integer product = element1 * element2;
        if sub_op then
            Elem[result, e, esize] = Elem[operand3, e, esize] - product;
        else
            Elem[result, e, esize] = Elem[operand3, e, esize] + product;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

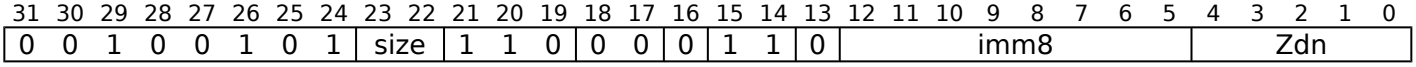
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MUL (immediate)

Multiply by immediate (unpredicated)

Multiply by an immediate each element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a signed 8-bit value in the range -128 to +127, inclusive. This instruction is unpredicated.



MUL <Zdn>.<T>, <Zdn>.<T>, #<imm>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = SInt(imm8);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<imm> Is the signed immediate operand, in the range -128 to 127, encoded in the "imm8" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    Elem[result, e, esize] = (element1 * imm)<esize-1:0>;

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

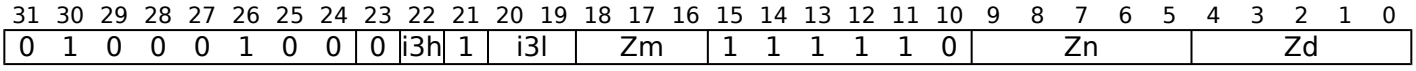
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

MUL (indexed)

Multiply (indexed)

Multiply all integer elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment. The results are placed in the corresponding elements of the destination vector.
The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element. This instruction is unpredicated.
It has encodings from 3 classes: 16-bit , 32-bit and 64-bit

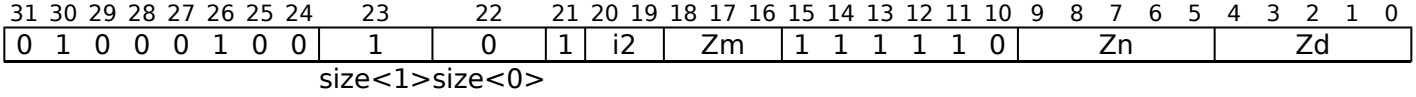
16-bit



MUL <Zd>.H, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

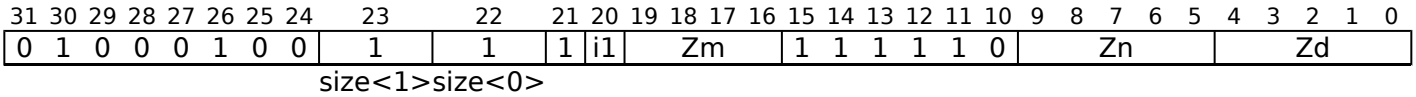
32-bit



MUL <Zd>.S, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

64-bit



MUL <Zd>.D, <Zn>.D, <Zm>.D[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the 16-bit and 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the 16-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields.

For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.

For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, s, esize]);
    integer res = element1 * element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

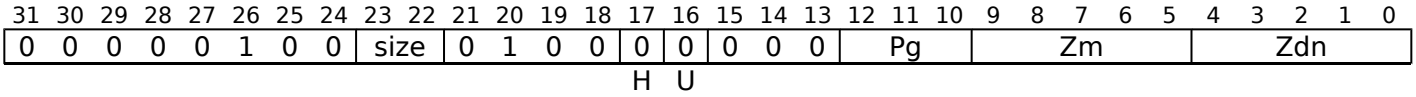
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MUL (vectors, predicated)

Multiply vectors (predicated)

Multiply active elements of the first source vector by corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



MUL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    if ElemP[mask, e, esize] == '1' then
        integer product = element1 * element2;
        Elem[result, e, esize] = product<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

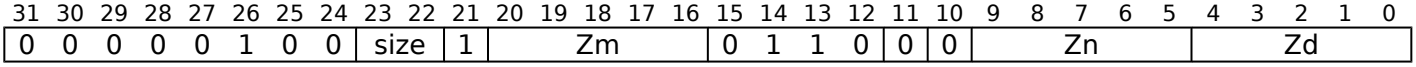
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MUL (vectors, unpredicated)

Multiply vectors (unpredicated)

Multiply all elements of the first source vector by corresponding elements of the second source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.



MUL <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer product = element1 * element2;
    Elem[result, e, esize] = product<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

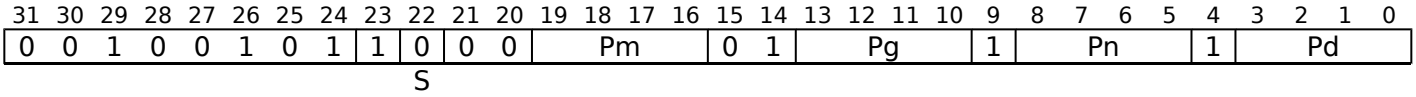
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

NAND

Bitwise NAND predicates

Bitwise NAND active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.



NAND <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm>	Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = NOT(element1 AND element2);
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

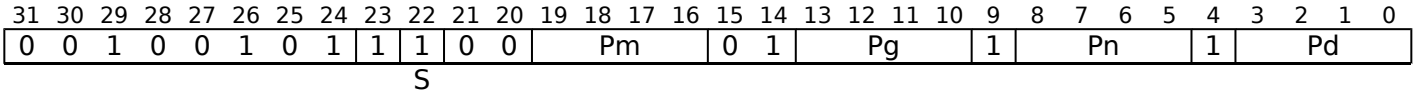
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

- The values of the NZCV flags.

NANDS

Bitwise NAND predicates, setting the condition flags

Bitwise NAND active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.



NANDS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm>	Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = NOT(element1 AND element2);
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

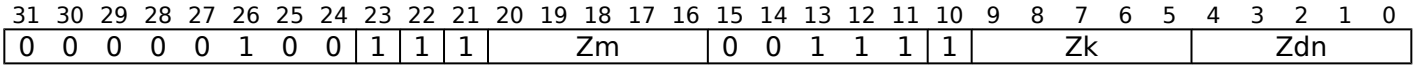
If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

NBSL

Bitwise inverted select

Selects bits from the first source vector where the corresponding bit in the third source vector is '1', and from the second source vector where the corresponding bit in the third source vector is '0'. The inverted result is placed destructively in the destination and first source vector. This instruction is unpredicated.



NBSL <Zdn>.D, <Zdn>.D, <Zm>.D, <Zk>.D

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
integer m = UInt(Zm);
integer k = UInt(Zk);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn>	Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.
<Zk>	Is the name of the third source scalable vector register, encoded in the "Zk" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[k, VL];

Z[dn, VL] = NOT((operand1 AND operand3) OR (operand2 AND NOT(operand3)));
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

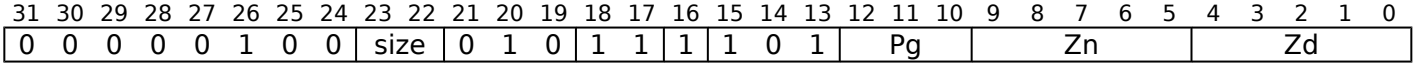
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

NEG

Negate (predicated)

Negate the signed integer value in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.



NEG <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element = SInt(Elem[operand, e, esize]);
        element = -element;
        Elem[result, e, esize] = element<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

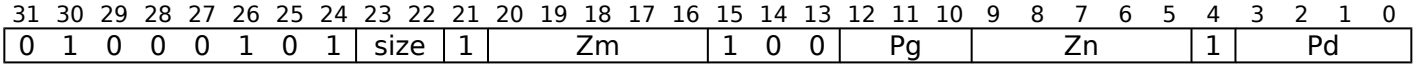
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

NMATCH

Detect no matching elements, setting the condition flags

This instruction compares each active 8-bit or 16-bit character in the first source vector with all of the characters in the corresponding 128-bit segment of the second source vector. Where the first source element detects no matching characters in the second segment it places true in the corresponding element of the destination predicate, otherwise false. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.



NMATCH <Pd>.<T>, <Pg>/Z, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE2() then UNDEFINED;
if size IN {'1x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer d = UInt(Pd);
integer n = UInt(Zn);
integer m = UInt(Zm);
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size<0>”:

size<0>	<T>
0	B
1	H

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

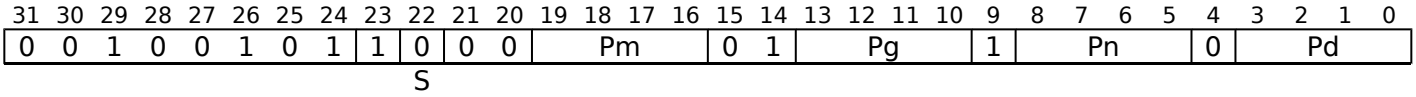
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(PL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer segmentbase = e - (e MOD eltspersegment);
        ElemP[result, e, esize] = '1';
        bits(esize) element1 = Elem[operand1, e, esize];
        for i = segmentbase to (segmentbase + eltspersegment) - 1
            bits(esize) element2 = Elem[operand2, i, esize];
            if element1 == element2 then
                ElemP[result, e, esize] = '0';
    else
        ElemP[result, e, esize] = '0';

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

NOR

Bitwise NOR predicates

Bitwise NOR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.



NOR <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm>	Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = NOT(element1 OR element2);
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

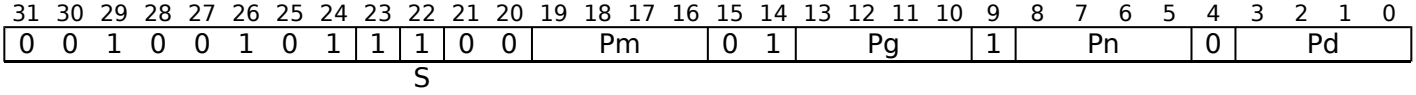
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

- The values of the NZCV flags.

NORS

Bitwise NOR predicates, setting the condition flags

Bitwise NOR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.



NORS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm>	Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = NOT(element1 OR element2);
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

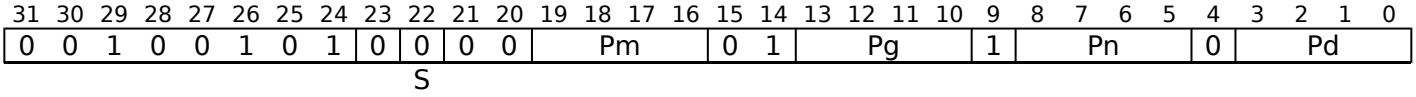
NOT (predicate)

Bitwise invert predicate

Bitwise invert each active element of the source predicate, and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This is an alias of EOR (predicates). This means:

- The encodings in this description are named to match the encodings of EOR (predicates).
- The description of EOR (predicates) gives the operational pseudocode for this instruction.



NOT <Pd>.B, <Pg>/Z, <Pn>.B

is equivalent to

EOR <Pd>.B, <Pg>/Z, <Pn>.B, <Pg>.B

and is the preferred disassembly when `Pm == Pg`.

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

The description of EOR (predicates) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

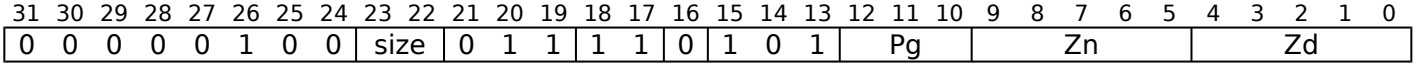
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

NOT (vector)

Bitwise invert vector (predicated)

Bitwise invert each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.



NOT <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = NOT element;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

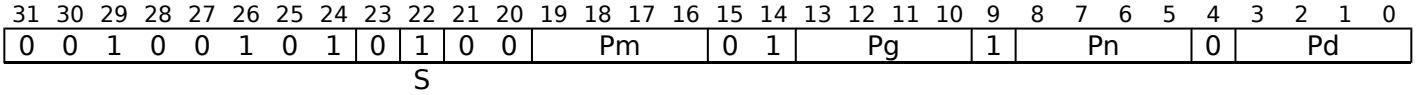
NOTS

Bitwise invert predicate, setting the condition flags

Bitwise invert each active element of the source predicate, and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.

This is an alias of EORS. This means:

- The encodings in this description are named to match the encodings of EORS.
- The description of EORS gives the operational pseudocode for this instruction.



NOTS <Pd>.B, <Pg>/Z, <Pn>.B

is equivalent to

EORS <Pd>.B, <Pg>/Z, <Pn>.B, <Pg>.B

and is the preferred disassembly when `Pm == Pg`.

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.

Operation

The description of EORS gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

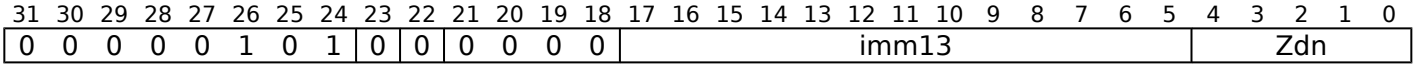
ORN (immediate)

Bitwise inclusive OR with inverted immediate (unpredicated)

Bitwise inclusive OR an inverted immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This is a pseudo-instruction of ORR (immediate). This means:

- The encodings in this description are named to match the encodings of ORR (immediate).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of ORR (immediate) gives the operational pseudocode for this instruction.



ORN <Zdn>.<T>, <Zdn>.<T>, #<const>

is equivalent to

ORR <Zdn>.<T>, <Zdn>.<T>, #(-<const> - 1)

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “imm13<12>:imm13<5:0>”:

imm13<12>	imm13<5:0>	<T>
0	0xxxxx	S
0	10xxxx	H
0	110xxx	B
0	1110xx	B
0	11110x	B
0	111110	RESERVED
0	111111	RESERVED
1	xxxxxx	D

<const> Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the "imm13" field.

Operation

The description of ORR (immediate) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

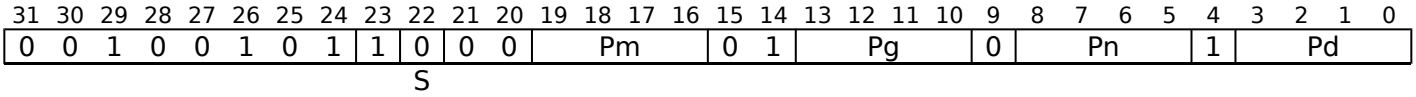
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ORN (predicates)

Bitwise inclusive OR inverted predicate

Bitwise inclusive OR inverted active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.



ORN <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm>	Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 OR (NOT element2);
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

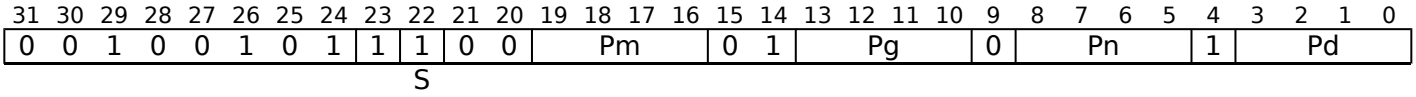
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

- The values of the NZCV flags.

ORNS

Bitwise inclusive OR inverted predicate, setting the condition flags

Bitwise inclusive OR inverted active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.



ORNS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm>	Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 OR (NOT element2);
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
- The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

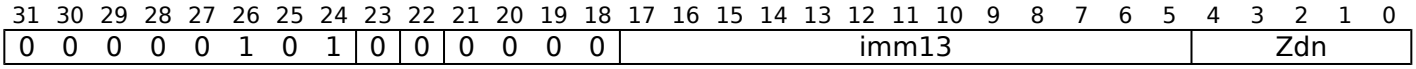
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ORR (immediate)

Bitwise inclusive OR with immediate (unpredicated)

Bitwise inclusive OR an immediate with each 64-bit element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a 64-bit value consisting of a single run of ones or zeros repeating every 2, 4, 8, 16, 32 or 64 bits. This instruction is unpredicated.

This instruction is used by the pseudo-instruction ORN (immediate).



ORR <Zdn>.<T>, <Zdn>.<T>, #<const>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer dn = UInt(Zdn);
bits(64) imm;
(imm, -) = DecodeBitMasks(imm13<12>, imm13<5:0>, imm13<11:6>, TRUE, 64);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “imm13<12>:imm13<5:0>”:

imm13<12>	imm13<5:0>	<T>
0	0xxxxx	S
0	10xxxx	H
0	110xxx	B
0	1110xx	B
0	11110x	B
0	111110	RESERVED
0	111111	RESERVED
1	xxxxxx	D

<const> Is a 64, 32, 16 or 8-bit bitmask consisting of replicated 2, 4, 8, 16, 32 or 64 bit fields, each field containing a rotated run of non-zero bits, encoded in the "imm13" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV 64;
bits(VL) operand = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(64) element1 = Elem[operand, e, 64];
    Elem[result, e, 64] = element1 OR imm;

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

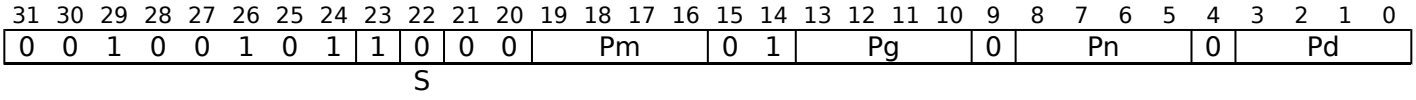
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ORR (predicates)

Bitwise inclusive OR predicates

Bitwise inclusive OR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.
This instruction is used by the alias MOV.



ORR <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm>	Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Alias Conditions

Alias	Is preferred when
MOV	`S == '0' && Pn == Pm && Pm == Pg`

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 OR element2;
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

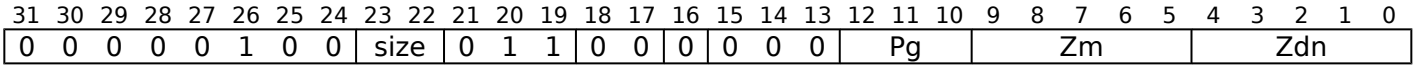
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ORR (vectors, predicated)

Bitwise inclusive OR vectors (predicated)

Bitwise inclusive OR active elements of the second source vector with corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



ORR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = element1 OR element2;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

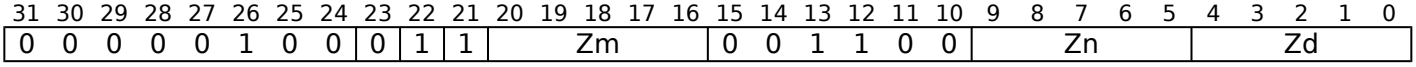
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ORR (vectors, unpredicated)

Bitwise inclusive OR vectors (unpredicated)

Bitwise inclusive OR all elements of the second source vector with corresponding elements of the first source vector and place the first in the corresponding elements of the destination vector. This instruction is unpredicated.

This instruction is used by the alias MOV (vector, unpredicated).



ORR <Zd>.D, <Zn>.D, <Zm>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Alias Conditions

Alias	Is preferred when
MOV (vector, unpredicated)	`Zn == Zm`

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];

Z[d, VL] = operand1 OR operand2;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

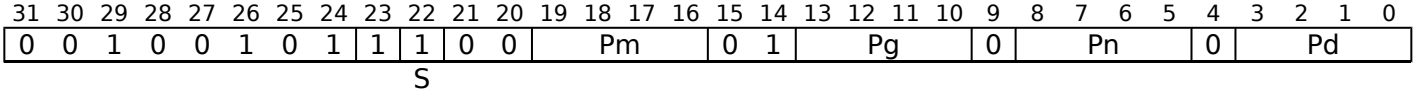
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ORRS

Bitwise inclusive OR predicates, setting the condition flags

Bitwise inclusive OR active elements of the second source predicate with corresponding elements of the first source predicate and place the results in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.
This instruction is used by the alias MOVS (unpredicated).



ORRS <Pd>.B, <Pg>/Z, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm>	Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Alias Conditions

Alias	Is preferred when
MOVS (unpredicated)	`S == '1' && Pn == Pm && Pm == Pg`

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1 OR element2;
    else
        ElemP[result, e, esize] = '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

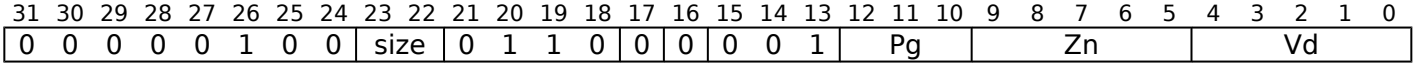
If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ORV

Bitwise inclusive OR reduction to scalar

Bitwise inclusive OR horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as zero.



ORV <V><d>, <Pg>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(esize) result = Zeros(esize);

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        result = result OR Elem[operand, e, esize];

V[d, esize] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

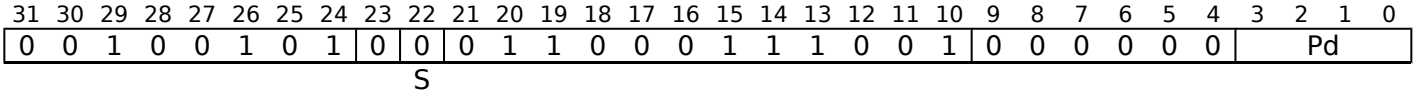
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PFALSE

Set all predicate elements to false

Set all elements in the destination predicate to false.



PFALSE <Pd>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer d = UInt(Pd);
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
P[d, PL] = Zeros(PL);
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

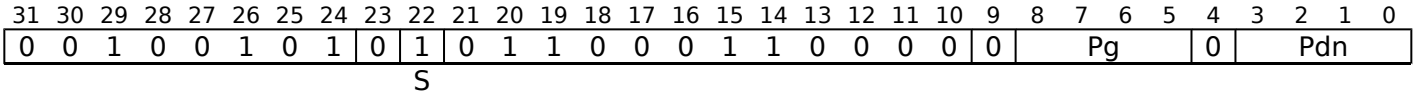
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PFIRST

Set the first active predicate element to true

Sets the first active element in the destination predicate to true, otherwise elements from the source predicate are passed through unchanged. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.



PFIRST <Pdn>.B, <Pg>, <Pdn>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer dn = UInt(Pdn);
```

Assembler Symbols

<Pdn>	Is the name of the source and destination scalable predicate register, encoded in the "Pdn" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) result = P[dn, PL];
integer first = -1;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' && first == -1 then
        first = e;

if first >= 0 then
    ElemP[result, first, esize] = '1';

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[dn, PL] = result;
```

Operational information

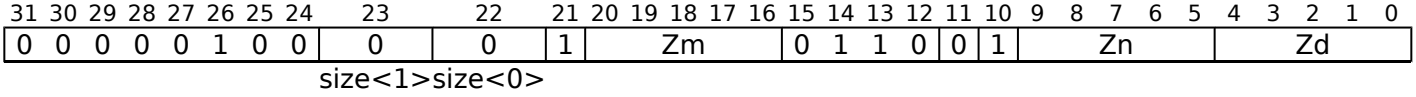
If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PMUL

Polynomial multiply vectors (unpredicated)

Polynomial multiply over [0, 1] all elements of the second source vector to corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.



PMUL <Zd>.B, <Zn>.B, <Zm>.B

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = PolynomialMult(element1, element2)<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

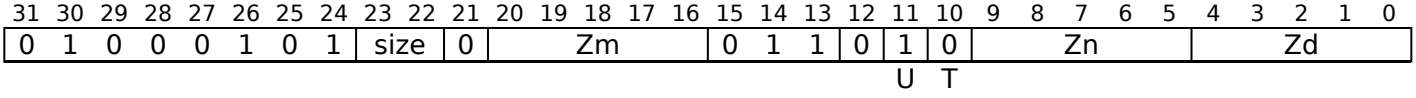
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PMULLB

Polynomial multiply long (bottom)

Polynomial multiply over [0, 1] the corresponding even-numbered elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.AES indicates whether the instruction variant with 64-bit source and 128-bit destination elements is implemented.



PMULLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' && !HaveSVE2PMULL128() then UNDEFINED;
integer esize;
case size of
    when '00' esize = 128;
    when '01' esize = 16;
    when '10' UNDEFINED;
    when '11' esize = 64;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	Q
01	H
10	RESERVED
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	D
01	B
10	RESERVED
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
if esize < 128 then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize DIV 2) element1 = Elem[operand1, 2*e + 0, esize DIV 2];
    bits(esize DIV 2) element2 = Elem[operand2, 2*e + 0, esize DIV 2];
    Elem[result, e, esize] = PolynomialMult(element1, element2);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

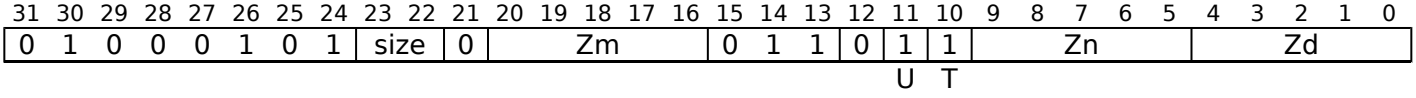
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PMULLT

Polynomial multiply long (top)

Polynomial multiply over [0, 1] the corresponding odd-numbered elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.AES indicates whether the instruction variant with 64-bit source and 128-bit destination elements is implemented.



PMULLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' && !HaveSVE2PMULL128() then UNDEFINED;
integer esize;
case size of
    when '00' esize = 128;
    when '01' esize = 16;
    when '10' UNDEFINED;
    when '11' esize = 64;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	Q
01	H
10	RESERVED
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	D
01	B
10	RESERVED
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
if esize < 128 then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize DIV 2) element1 = Elem[operand1, 2*e + 1, esize DIV 2];
    bits(esize DIV 2) element2 = Elem[operand2, 2*e + 1, esize DIV 2];
    Elem[result, e, esize] = PolynomialMult(element1, element2);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

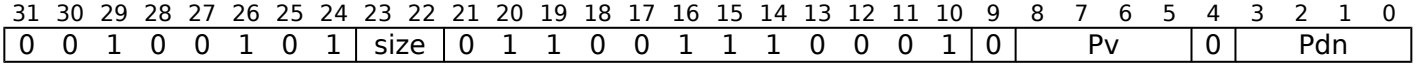
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PNEXT

Find next active predicate

An instruction used to construct a loop which iterates over all true elements in the vector select predicate register. If all elements in the first source predicate register are false it determines the first true element in the vector select predicate register, otherwise it determines the next true element in the vector select predicate register that follows the last true element in the first source predicate register. All elements of the destination predicate register are set to false, except the element corresponding to the determined vector select element, if any, which is set to true. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.



PNEXT <Pdn>.<T>, <Pv>, <Pdn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer v = UInt(Pv);
integer dn = UInt(Pdn);
```

Assembler Symbols

<Pdn> Is the name of the first source and destination scalable predicate register, encoded in the "Pdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pv> Is the name of the vector select predicate register, encoded in the "Pv" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[v, PL];
bits(PL) operand = P[dn, PL];
bits(PL) result;

integer next = LastActiveElement(operand, esize) + 1;

while next < elements && (ElemP[mask, next, esize] == '0') do
    next = next + 1;

result = Zeros(PL);
if next < elements then
    ElemP[result, next, esize] = '1';

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[dn, PL] = result;
```

Operational information

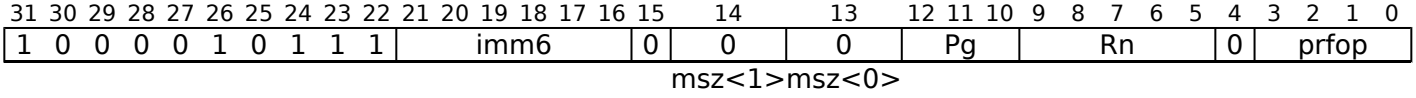
If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PRFB (scalar plus immediate)

Contiguous prefetch bytes (immediate index)

Contiguous prefetch of byte elements from the memory address generated by a 64-bit scalar base and immediate index in the range -32 to 31 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.
The predicate may be used to suppress prefetches from unwanted addresses.



PRFB <prfop>, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 0;
integer offset = SInt(imm6);
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -32 to 31, defaulting to 0, encoded in the "imm6" field.

Operation

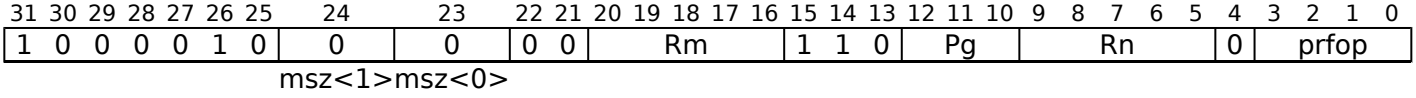
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + (eoff << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PRFB (scalar plus scalar)

Contiguous prefetch bytes (scalar index)

Contiguous prefetch of byte elements from the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element prefetch the index value is incremented, but the index register is not updated.
The predicate may be used to suppress prefetches from unwanted addresses.



PRFB <prfop>, <Pg>, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 0;
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(64) offset;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + (eoff << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```

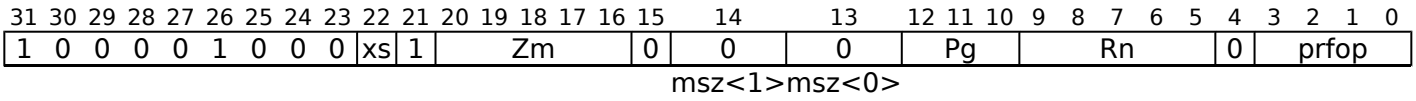
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PRFB (scalar plus vector)

Gather prefetch bytes (scalar plus vector)

Gather prefetch of bytes from the active memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive addresses are not prefetched from memory.
The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 3 classes: 32-bit scaled offset , 32-bit unpacked scaled offset and 64-bit scaled offset

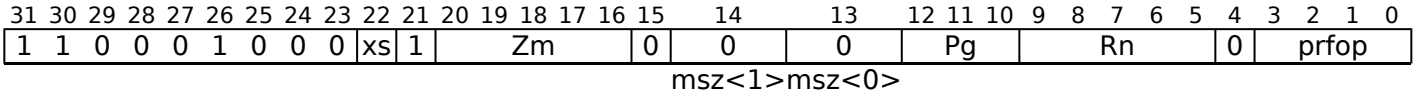
32-bit scaled offset



PRFB <prfop>, <Pg>, [<Xn|SP>, <Zm>.S, <mod>]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 0;
```

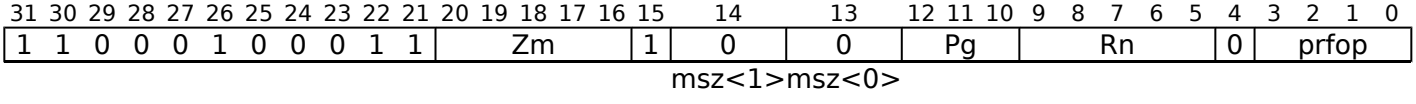
32-bit unpacked scaled offset



PRFB <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, <mod>]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 0;
```

64-bit scaled offset



PRFB <prfop>, <Pg>, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(VL) offset;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```

PRFB (vector plus immediate)

Gather prefetch bytes (vector plus immediate)

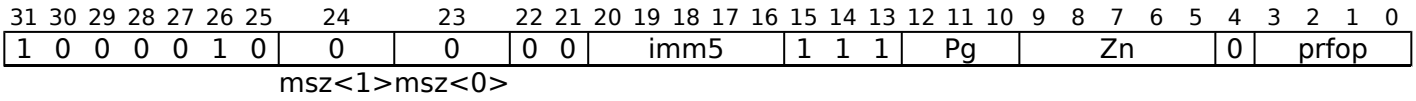
Gather prefetch of bytes from the active memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive addresses are not prefetched from memory.

The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

It has encodings from 2 classes: 32-bit element and 64-bit element

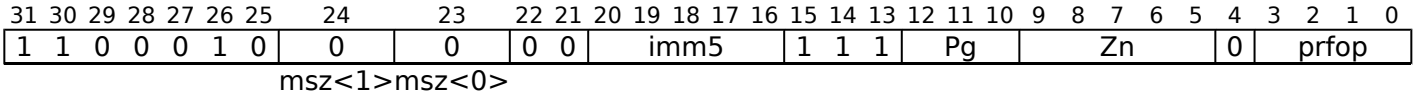
32-bit element



PRFB <prfop>, <Pg>, [<Zn>.S{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 0;
integer offset = UInt(imm5);
```

64-bit element



PRFB <prfop>, <Pg>, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 0;
integer offset = UInt(imm5);
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm> Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.

Operation

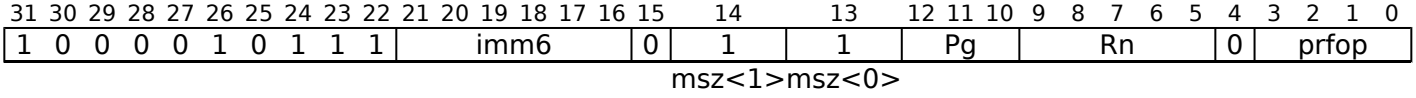
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + (offset << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PRFD (scalar plus immediate)

Contiguous prefetch doublewords (immediate index)

Contiguous prefetch of doubleword elements from the memory address generated by a 64-bit scalar base and immediate index in the range -32 to 31 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.
The predicate may be used to suppress prefetches from unwanted addresses.



PRFD <prfop>, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 3;
integer offset = SInt(imm6);
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -32 to 31, defaulting to 0, encoded in the "imm6" field.

Operation

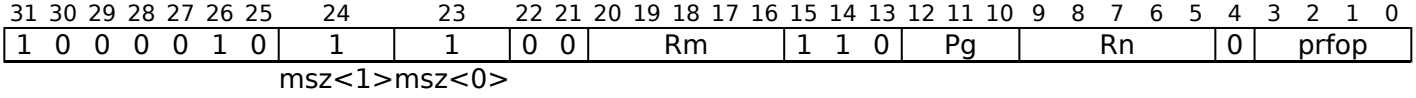
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + (eoff << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PRFD (scalar plus scalar)

Contiguous prefetch doublewords (scalar index)

Contiguous prefetch of doubleword elements from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element prefetch the index value is incremented, but the index register is not updated.
The predicate may be used to suppress prefetches from unwanted addresses.



PRFD <prfop>, <Pg>, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 3;
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(64) offset;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + (eoff << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```

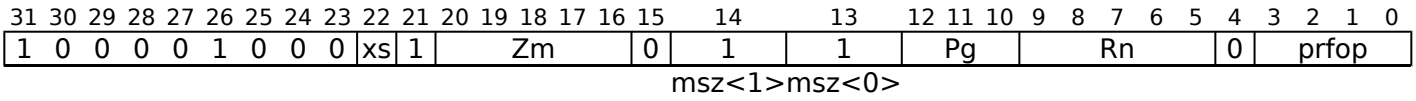
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PRFD (scalar plus vector)

Gather prefetch doublewords (scalar plus vector)

Gather prefetch of doublewords from the active memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then multiplied by 8. Inactive addresses are not prefetched from memory.
The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 3 classes: 32-bit scaled offset , 32-bit unpacked scaled offset and 64-bit scaled offset

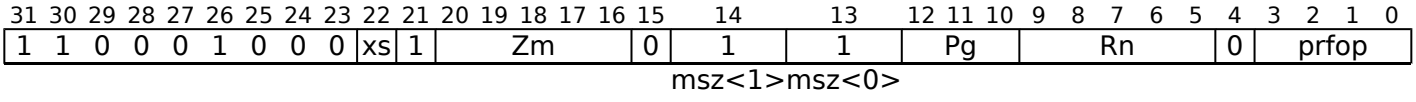
32-bit scaled offset



PRFD <prfop>, <Pg>, [<Xn|SP>, <Zm>.S, <mod> #3]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 3;
```

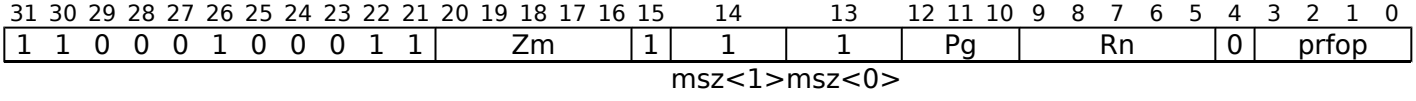
32-bit unpacked scaled offset



PRFD <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #3]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 3;
```

64-bit scaled offset



PRFD <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, LSL #3]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 3;
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(VL) offset;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```

PRFD (vector plus immediate)

Gather prefetch doublewords (vector plus immediate)

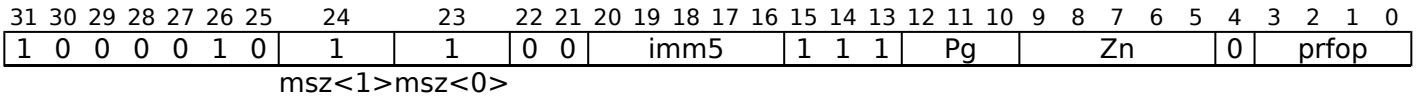
Gather prefetch of doublewords from the active memory addresses generated by a vector base plus immediate index. The index is a multiple of 8 in the range 0 to 248. Inactive addresses are not prefetched from memory.

The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

It has encodings from 2 classes: 32-bit element and 64-bit element

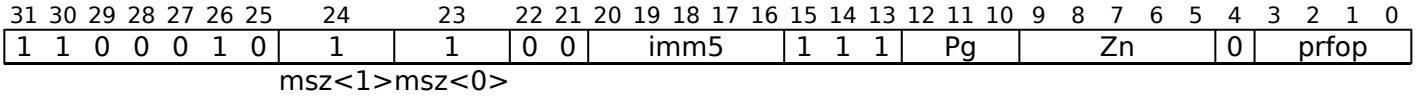
32-bit element



PRFD <prfop>, <Pg>, [<Zn>.S{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 3;
integer offset = UInt(imm5);
```

64-bit element



PRFD <prfop>, <Pg>, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 3;
integer offset = UInt(imm5);
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm> Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 248, defaulting to 0, encoded in the "imm5" field.

Operation

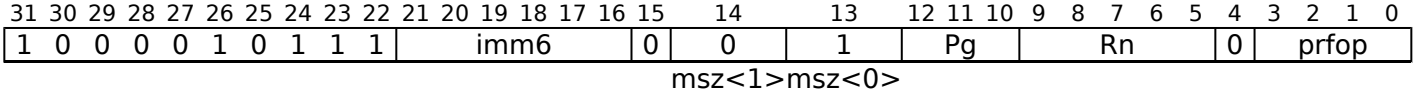
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + (offset << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PRFH (scalar plus immediate)

Contiguous prefetch halfwords (immediate index)

Contiguous prefetch of halfword elements from the memory address generated by a 64-bit scalar base and immediate index in the range -32 to 31 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.
The predicate may be used to suppress prefetches from unwanted addresses.



PRFH <prfop>, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 1;
integer offset = SInt(imm6);
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -32 to 31, defaulting to 0, encoded in the "imm6" field.

Operation

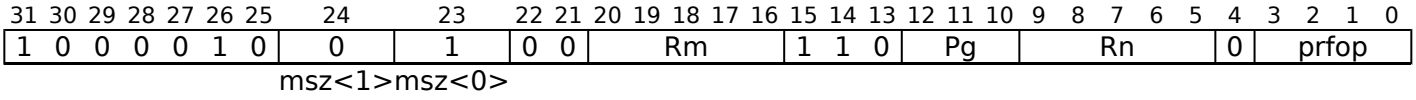
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + (eoff << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PRFH (scalar plus scalar)

Contiguous prefetch halfwords (scalar index)

Contiguous prefetch of halfword elements from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element prefetch the index value is incremented, but the index register is not updated.
The predicate may be used to suppress prefetches from unwanted addresses.



PRFH <prfop>, <Pg>, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
constant integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 1;
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(64) offset;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + (eoff << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```

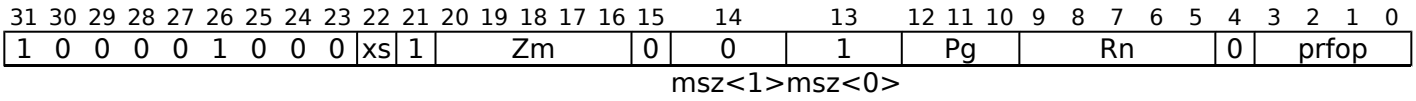
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PRFH (scalar plus vector)

Gather prefetch halfwords (scalar plus vector)

Gather prefetch of halfwords from the active memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then multiplied by 2. Inactive addresses are not prefetched from memory.
The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 3 classes: 32-bit scaled offset , 32-bit unpacked scaled offset and 64-bit scaled offset

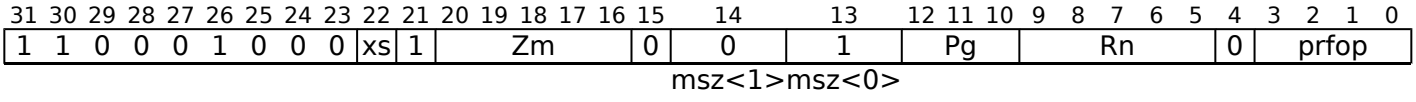
32-bit scaled offset



PRFH <prfop>, <Pg>, [<Xn|SP>, <Zm>.S, <mod> #1]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 1;
```

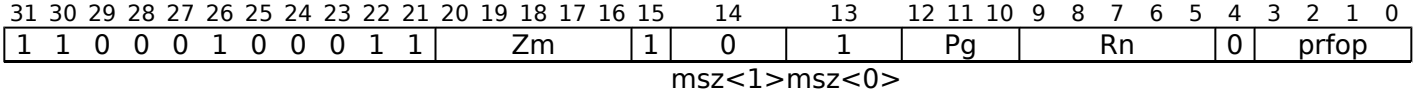
32-bit unpacked scaled offset



PRFH <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #1]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 1;
```

64-bit scaled offset



PRFH <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, LSL #1]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 1;
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(VL) offset;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```

PRFH (vector plus immediate)

Gather prefetch halfwords (vector plus immediate)

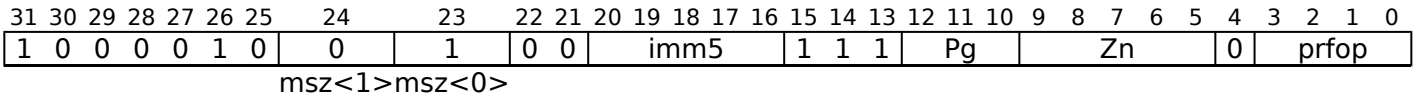
Gather prefetch of halfwords from the active memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive addresses are not prefetched from memory.

The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

It has encodings from 2 classes: 32-bit element and 64-bit element

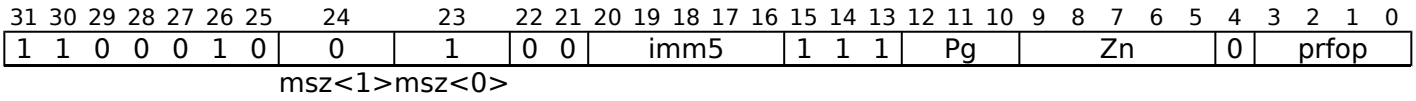
32-bit element



PRFH <prfop>, <Pg>, [<Zn>.S{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 1;
integer offset = UInt(imm5);
```

64-bit element



PRFH <prfop>, <Pg>, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 1;
integer offset = UInt(imm5);
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm> Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the "imm5" field.

Operation

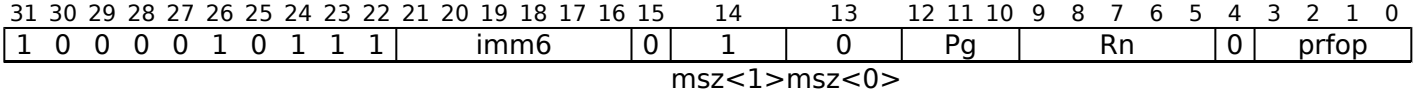
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + (offset << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PRFW (scalar plus immediate)

Contiguous prefetch words (immediate index)

Contiguous prefetch of word elements from the memory address generated by a 64-bit scalar base and immediate index in the range -32 to 31 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address.
The predicate may be used to suppress prefetches from unwanted addresses.



PRFW <prfop>, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 2;
integer offset = SInt(imm6);
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -32 to 31, defaulting to 0, encoded in the "imm6" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + (eoff << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```

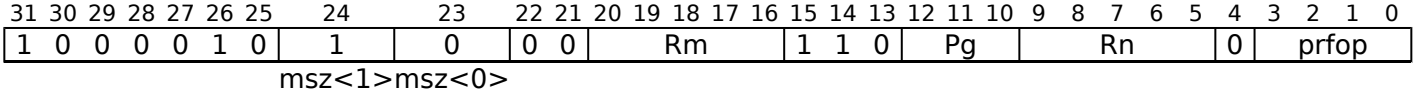
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PRFW (scalar plus scalar)

Contiguous prefetch words (scalar index)

Contiguous prefetch of word elements from the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element prefetch the index value is incremented, but the index register is not updated.

The predicate may be used to suppress prefetches from unwanted addresses.



PRFW <prfop>, <Pg>, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 2;
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(64) offset;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = UInt(offset) + e;
        bits(64) addr = base + (eoff << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```

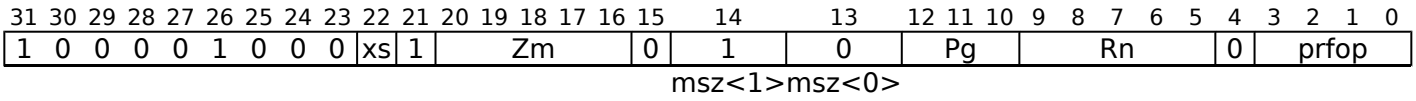
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PRFW (scalar plus vector)

Gather prefetch words (scalar plus vector)

Gather prefetch of words from the active memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then multiplied by 4. Inactive addresses are not prefetched from memory.
The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 3 classes: 32-bit scaled offset , 32-bit unpacked scaled offset and 64-bit scaled offset

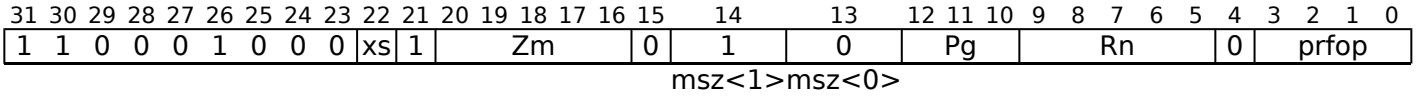
32-bit scaled offset



PRFW <prfop>, <Pg>, [<Xn|SP>, <Zm>.S, <mod> #2]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 2;
```

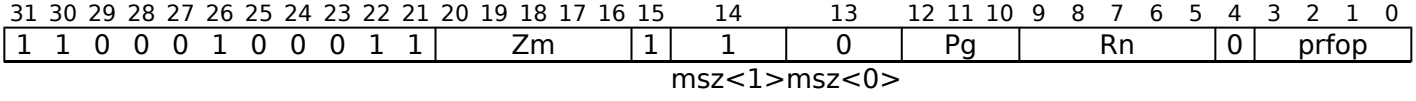
32-bit unpacked scaled offset



PRFW <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #2]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 32;
boolean offs_unsigned = (xs == '0');
integer scale = 2;
```

64-bit scaled offset



PRFW <prfop>, <Pg>, [<Xn|SP>, <Zm>.D, LSL #2]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
constant integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 2;
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(64) base;
bits(VL) offset;

if AnyActiveElement(mask, esize) then
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```

PRFW (vector plus immediate)

Gather prefetch words (vector plus immediate)

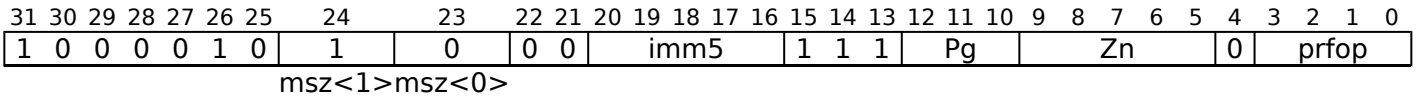
Gather prefetch of words from the active memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive addresses are not prefetched from memory.

The <prfop> symbol specifies the prefetch hint as a combination of three options: access type PLD for load or PST for store; target cache level L1, L2 or L3; temporality (KEEP for temporal or STRM for non-temporal).

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

It has encodings from 2 classes: 32-bit element and 64-bit element

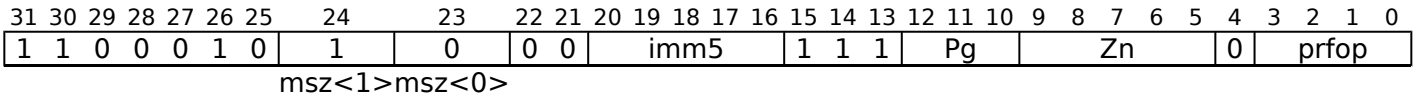
32-bit element



PRFW <prfop>, <Pg>, [<Zn>.S{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 2;
integer offset = UInt(imm5);
```

64-bit element



PRFW <prfop>, <Pg>, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer level = UInt(prfop<2:1>);
boolean stream = (prfop<0> == '1');
pref_hint = if prfop<3> == '0' then Prefetch_READ else Prefetch_WRITE;
integer scale = 2;
integer offset = UInt(imm5);
```

Assembler Symbols

<prfop> Is the prefetch operation specifier, encoded in “prfop”:

prfop	<prfop>
0000	PLDL1KEEP
0001	PLDL1STRM
0010	PLDL2KEEP
0011	PLDL2STRM
0100	PLDL3KEEP
0101	PLDL3STRM
x11x	#uimm4
1000	PSTL1KEEP
1001	PSTL1STRM
1010	PSTL2KEEP
1011	PSTL2STRM
1100	PSTL3KEEP
1101	PSTL3STRM

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the base scalable vector register, encoded in the "Zn" field.

<imm> Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the "imm5" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + (offset << scale);
        Hint_Prefetch(addr, pref_hint, level, stream);
```

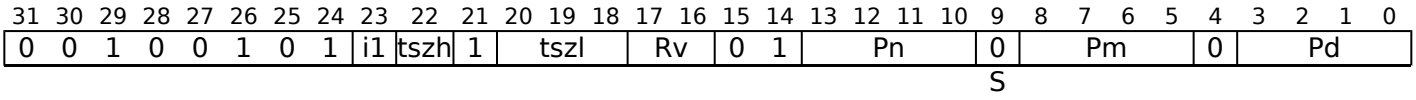
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PSEL

Predicate select between predicate register or all-false

If the indexed element of the second source predicate is true, place the contents of the first source predicate register into the destination predicate register, otherwise set the destination predicate to all-false. The indexed element is determined by the sum of a general-purpose index register and an immediate, modulo the number of elements. Does not set the condition flags.

SVE2
(FEAT_SME)



PSEL <Pd>, <Pn>, <Pm>.<T>[<Wv>, <imm>]

```
if !HaveSME() then UNDEFINED;
bits(5) imm5 = i1:tszh:tszl;
integer esize;
integer imm;
case tszh:tszl of
    when '0000' UNDEFINED;
    when '1000' esize = 64;  imm = UInt(imm5<4>);
    when 'x100' esize = 32;  imm = UInt(imm5<4:3>);
    when 'xx10' esize = 16;  imm = UInt(imm5<4:2>);
    when 'xxx1' esize = 8;   imm = UInt(imm5<4:1>);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
integer v = UInt('011':Rv);
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	000	RESERVED
x	xx1	B
x	x10	H
x	100	S
1	000	D

<Wv> Is the 32-bit name of the vector select register W12-W15, encoded in the "Rv" field.

<imm> Is the element index, in the range 0 to one less than the number of vector elements in a 128-bit vector register, encoded in "i1:tszh:tszl".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(32) idx = X[v, 32];
integer element = (UInt(idx) + imm) MOD elements;
bits(PL) result;

if ElemP[operand2, element, esize] == '1' then
    result = operand1;
else
    result = Zeros(PL);

P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

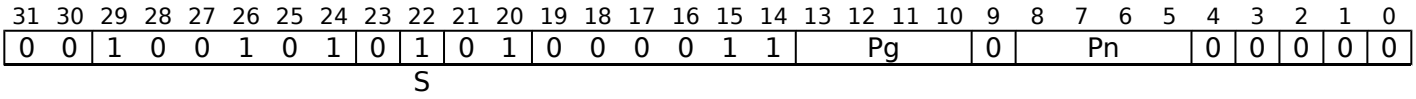
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PTEST

Set condition flags for predicate

Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate source register, and the V flag to zero.



PTEST <Pg>, <Pn>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
```

Assembler Symbols

<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the source scalable predicate register, encoded in the "Pn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) mask = P[g, PL];
bits(PL) result = P[n, PL];

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the predicate register written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PTRUE

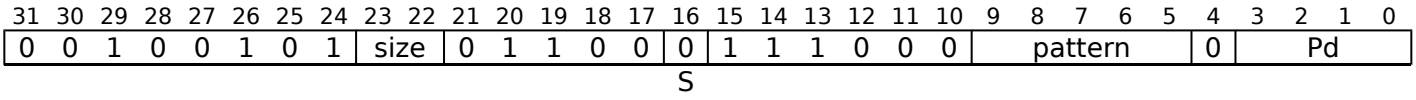
Initialise predicate from named constraint

Set elements of the destination predicate to true if the element number satisfies the named predicate constraint, or to false otherwise. If the constraint specifies more elements than are available at the current vector length then all elements of the destination predicate are set to false.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception. Does not set the condition flags.



PTRUE <Pd>.<T>{, <pattern>}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer d = UInt(Pd);
boolean setflags = FALSE;
bits(5) pat = pattern;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(PL) result;

for e = 0 to elements-1
    ElemP[result, e, esize] = if e < count then '1' else '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(result, result, esize);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PTRUES

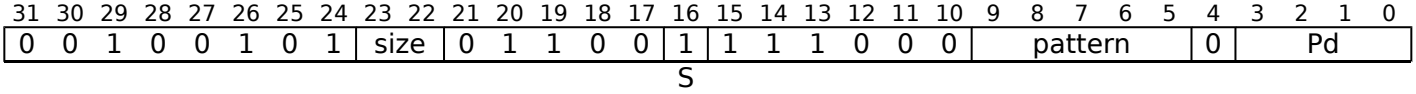
Initialise predicate from named constraint and set the condition flags

Set elements of the destination predicate to true if the element number satisfies the named predicate constraint, or to false otherwise. If the constraint specifies more elements than are available at the current vector length then all elements of the destination predicate are set to false.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.



PTRUES <Pd>.<T>{, <pattern>}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer d = UInt(Pd);
boolean setflags = TRUE;
bits(5) pat = pattern;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(PL) result;

for e = 0 to elements-1
    ElemP[result, e, esize] = if e < count then '1' else '0';

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(result, result, esize);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the NZCV condition flags written by this instruction might be significantly delayed.

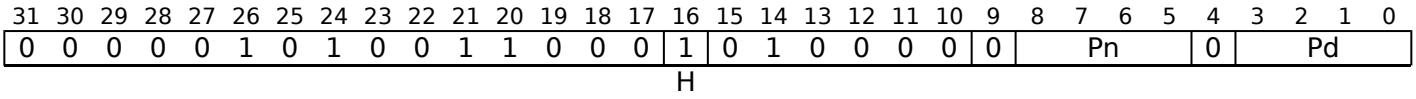
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

PUNPKHI, PUNPKLO

Unpack and widen half of predicate

Unpack elements from the lowest or highest half of the source predicate and place in elements of twice their size within the destination predicate. This instruction is unpredicated.
It has encodings from 2 classes: High half and Low half

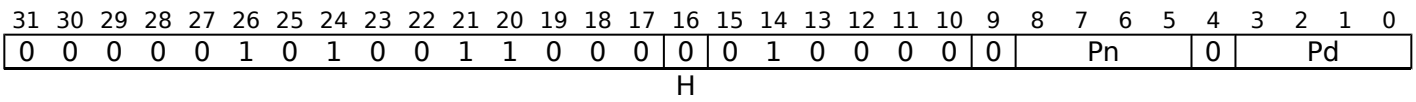
High half



PUNPKHI <Pd>.H, <Pn>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean hi = TRUE;
```

Low half



PUNPKLO <Pd>.H, <Pn>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer n = UInt(Pn);
integer d = UInt(Pd);
boolean hi = FALSE;
```

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pn>	Is the name of the source scalable predicate register, encoded in the "Pn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) operand = P[n, PL];
bits(PL) result;

for e = 0 to elements-1
    ElemP[result, e, esize] = ElemP[operand, if hi then e + elements else e, esize DIV 2];

P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.

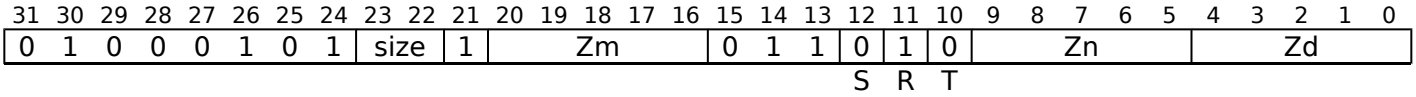
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

RADDHNB

Rounding add narrow high part (bottom)

Add each vector element of the first source vector to the corresponding vector element of the second source vector, and place the most significant rounded half of the result in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. This instruction is unpredicated.



RADDHNB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	B
10	H
11	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	H
10	S
11	D

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
constant integer halfesize = esize DIV 2;
constant integer round_const = 1 << (halfesize - 1);

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer res = ((element1 + element2) + round_const) >> halfesize;
    Elem[result, 2*e + 0, halfesize] = res<halfesize-1:0>;
    Elem[result, 2*e + 1, halfesize] = Zeros(halfesize);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

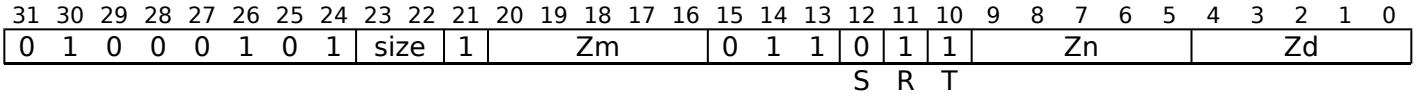
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

RADDHNT

Rounding add narrow high part (top)

Add each vector element of the first source vector to the corresponding vector element of the second source vector, and place the most significant rounded half of the result in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. This instruction is unpredicated.



RADDHNT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	B
10	H
11	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	H
10	S
11	D

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[d, VL];
constant integer halfesize = esize DIV 2;
constant integer round_const = 1 << (halfesize - 1);

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer res = ((element1 + element2) + round_const) >> halfesize;
    Elem[result, 2*e + 1, halfesize] = res<halfesize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

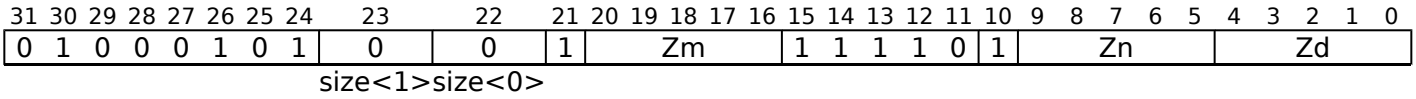
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

RAX1

Bitwise rotate left by 1 and exclusive OR

Rotate each 64-bit element of the second source vector left by 1 and exclusive OR with the corresponding elements of the first source vector. The results are placed in the corresponding elements of the destination vector. This instruction is unpredicated.
ID_AA64ZFR0_EL1.SHA3 indicates whether this instruction is implemented.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE2
(FEAT_SVE_SHA3)



RAX1 <Zd>.D, <Zn>.D, <Zm>.D

```
if !HaveSVE2SHA3() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 64;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(64) element1 = Elem[operand1, e, 64];
    bits(64) element2 = Elem[operand2, e, 64];
    Elem[result, e, 64] = element1 EOR ROL(element2, 1);
Z[d, VL] = result;
```

Operational information

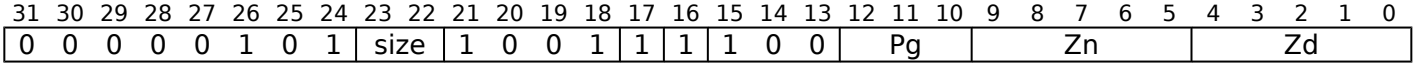
If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

RBIT

Reverse bits (predicated)

Reverse bits in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.



RBIT <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = BitReverse(element);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

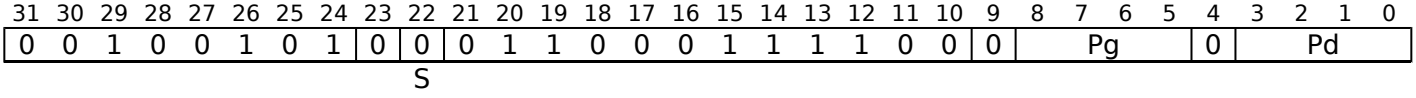
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

RDFFR (predicated)

Return predicate of succesfully loaded elements

Read the first-fault register (FFR) and place active elements in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Does not set the condition flags.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.



RDFFR <Pd>.B, <Pg>/Z

```
if !HaveSVE() then UNDEFINED;
integer g = UInt(Pg);
integer d = UInt(Pd);
boolean setflags = FALSE;
```

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) mask = P[g, PL];
bits(PL) ffr = FFR[PL];
bits(PL) result = ffr AND mask;

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, 8);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

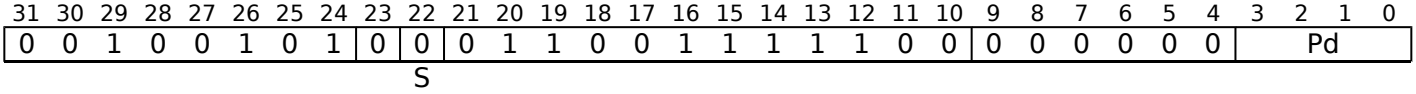
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

RDFFR (unpredicated)

Read the first-fault register

Read the first-fault register (FFR) and place in the destination predicate without predication.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.



RDFFR <Pd>.B

```
if !HaveSVE() then UNDEFINED;
integer d = UInt(Pd);
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) ffr = FFR[PL];
P[d, PL] = ffr;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

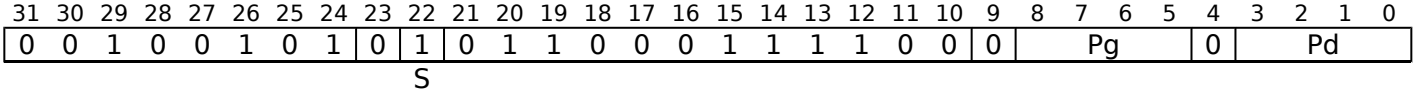
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

RDFFRS

Return predicate of succesfully loaded elements, setting the condition flags

Read the first-fault register (FFR) and place active elements in the corresponding elements of the destination predicate. Inactive elements in the destination predicate register are set to zero. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.



RDFFRS <Pd>.B, <Pg>/Z

```
if !HaveSVE() then UNDEFINED;
integer g = UInt(Pg);
integer d = UInt(Pd);
boolean setflags = TRUE;
```

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) mask = P[g, PL];
bits(PL) ffr = FFR[PL];
bits(PL) result = ffr AND mask;

if setflags then
    PSTATE.<N,Z,C,V> = PredTest(mask, result, 8);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

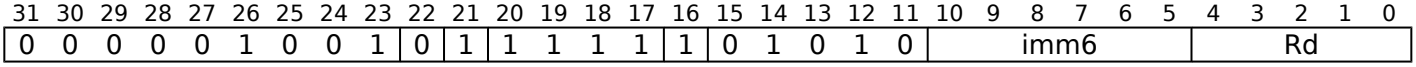
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

RDVL

Read multiple of vector register size to scalar register

Multiply the current vector register size in bytes by an immediate in the range -32 to 31 and place the result in the 64-bit destination general-purpose register.



RDVL <Xd>, #<imm>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer d = UInt(Rd);
integer imm = SInt(imm6);
```

Assembler Symbols

<Xd>	Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.
<imm>	Is the signed immediate operand, in the range -32 to 31, encoded in the "imm6" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
integer len = imm * (VL DIV 8);
X[d, 64] = len<63:0>;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

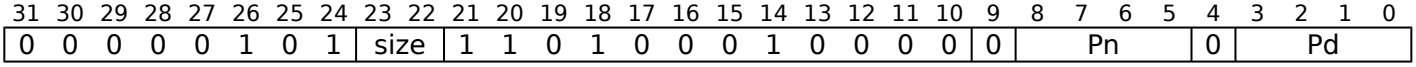
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

REV (predicate)

Reverse all elements in a predicate

Reverse the order of all elements in the source predicate and place in the destination predicate. This instruction is unpredicated.



REV <Pd>.<T>, <Pn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer d = UInt(Pd);
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) operand = P[n, PL];
bits(PL) result = Reverse(operand, esize DIV 8);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

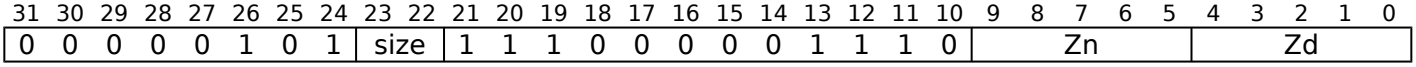
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

REV (vector)

Reverse all elements in a vector (unpredicated)

Reverse the order of all elements in the source vector and place in the destination vector. This instruction is unpredicated.



REV <Zd>.<T>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(VL) operand = Z[n, VL];
bits(VL) result = Reverse(operand, esize);
Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

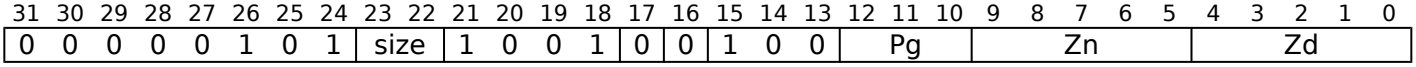
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

REVB, REVH, REVW

Reverse bytes / halfwords / words within elements (predicated)

Reverse the order of 8-bit bytes, 16-bit halfwords or 32-bit words within each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.
It has encodings from 3 classes: Byte , Halfword and Word

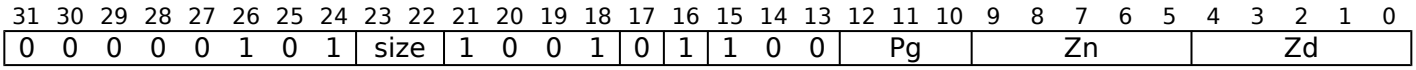
Byte



REVB <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer swsize = 8;
```

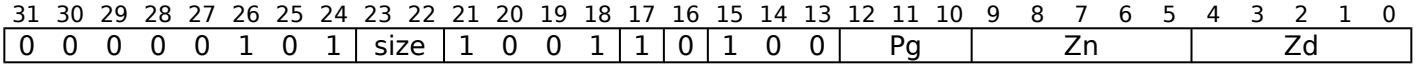
Halfword



REVH <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer swsize = 16;
```

Word



REVW <Zd>.D, <Pg>/M, <Zn>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size != '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer swsize = 32;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> For the byte variant: is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

For the halfword variant: is the size specifier, encoded in “size<0>”:

size<0>	<T>
0	S
1	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = Reverse(element, swsize);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

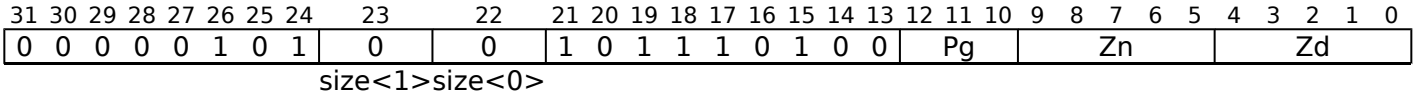
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

REVD

Reverse 64-bit doublewords in elements (predicated)

Reverse the order of 64-bit doublewords within each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.

SVE2
(FEAT_SME)



REVD <Zd>.Q, <Pg>/M, <Zn>.Q

```
if !HaveSME() then UNDEFINED;
constant integer esize = 128;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer swsize = 64;
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = Reverse(element, swsize);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

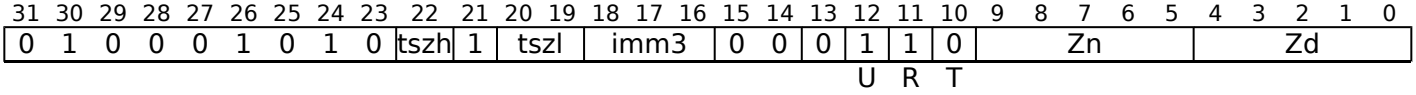
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

RSHRNB

Rounding shift right narrow by immediate (bottom)

Shift each unsigned integer value in the source vector elements right by an immediate value, and place the rounded results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



RSHRNB <Zd>.<T>, <Zn>.<Tb>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '000' UNDEFINED;
    when '001' esize = 8;
    when '01x' esize = 16;
    when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;
integer round_const = 1 << (shift-1);

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = (UInt(element) + round_const) >> shift;
    Elem[result, 2*e + 0, esize] = res<esize-1:0>;
    Elem[result, 2*e + 1, esize] = Zeros(esize);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

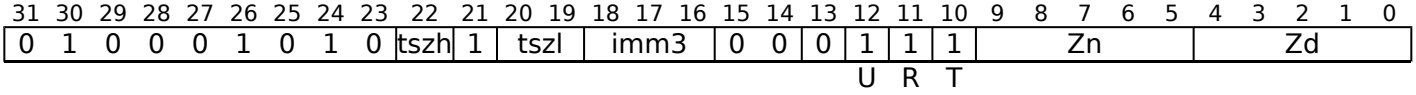
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

RSHRNT

Rounding shift right narrow by immediate (top)

Shift each unsigned integer value in the source vector elements right by an immediate value, and place the rounded results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



RSHRNT <Zd>.<T>, <Zn>.<Tb>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '000' UNDEFINED;
    when '001' esize = 8;
    when '01x' esize = 16;
    when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result = Z[d, VL];
integer round_const = 1 << (shift-1);

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = (UInt(element) + round_const) >> shift;
    Elem[result, 2*e + 1, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

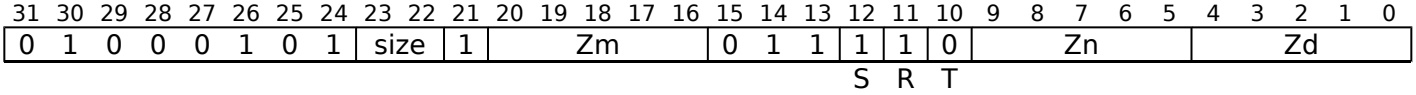
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

RSUBHNB

Rounding subtract narrow high part (bottom)

Subtract each vector element of the second source vector from the corresponding vector element in the first source vector, and place the most significant rounded half of the result in the even-numbered half-width destination elements, while setting the odd-numbered half-width destination elements to zero. This instruction is unpredicated.



RSUBHNB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	B
10	H
11	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	H
10	S
11	D

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
constant integer halfesize = esize DIV 2;
constant integer round_const = 1 << (halfesize - 1);

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer res = ((element1 - element2) + round_const) >> halfesize;
    Elem[result, 2*e + 0, halfesize] = res<halfesize-1:0>;
    Elem[result, 2*e + 1, halfesize] = Zeros(halfesize);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

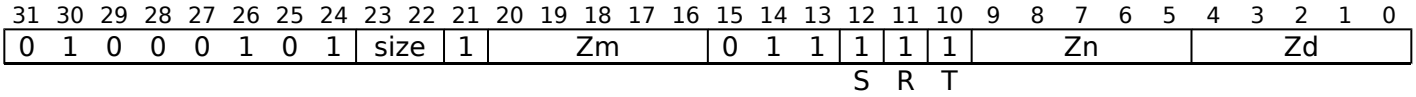
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

RSUBHNT

Rounding subtract narrow high part (top)

Subtract each vector element of the second source vector from the corresponding vector element in the first source vector, and place the most significant rounded half of the result in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. This instruction is unpredicated.



RSUBHNT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	B
10	H
11	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	H
10	S
11	D

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[d, VL];
constant integer halfesize = esize DIV 2;
constant integer round_const = 1 << (halfesize - 1);

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer res = ((element1 - element2) + round_const) >> halfesize;
    Elem[result, 2*e + 1, halfesize] = res<halfesize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

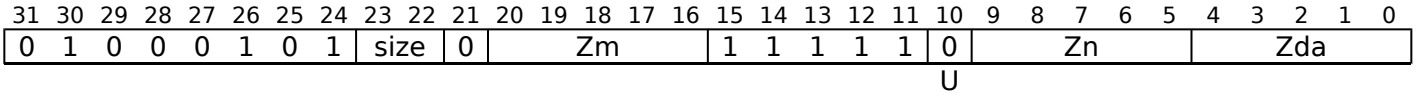
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SABA

Signed absolute difference and accumulate

Compute the absolute difference between signed integer values in elements of the second source vector and corresponding elements of the first source vector, and add the difference to the corresponding elements of the destination vector. This instruction is unpredicated.



SABA <Zda>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean unsigned = FALSE;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    bits(esize) absdiff = Abs(element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

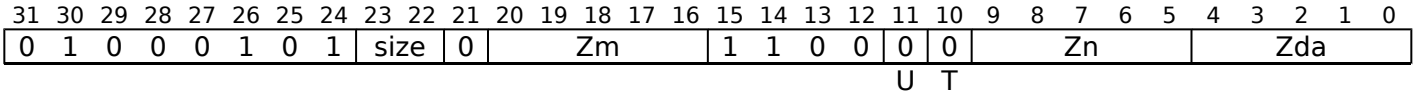
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SABALB

Signed absolute difference and accumulate long (bottom)

Compute the absolute difference between even-numbered signed integer values in elements of the second source vector and corresponding elements of the first source vector, and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.



SABALB <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2*e + 0, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    bits(esize) absdiff = Abs(element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

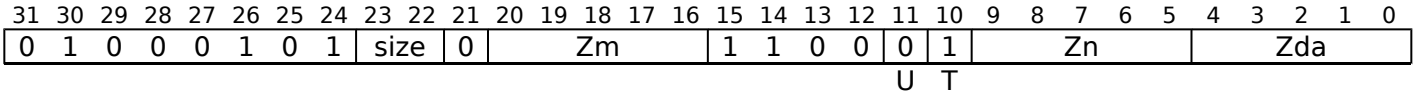
- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SABALT

Signed absolute difference and accumulate long (top)

Compute the absolute difference between odd-numbered signed elements of the second source vector and corresponding elements of the first source vector, and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.



SABALT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2*e + 1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    bits(esize) absdiff = Abs(element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

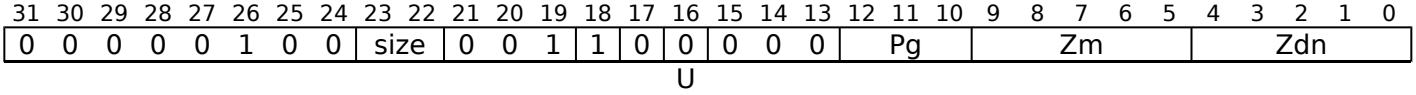
- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SABD

Signed absolute difference (predicated)

Compute the absolute difference between signed integer values in active elements of the second source vector and corresponding elements of the first source vector and destructively place the difference in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



SABD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    if ElemP[mask, e, esize] == '1' then
        integer absdiff = Abs(element1 - element2);
        Elem[result, e, esize] = absdiff<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

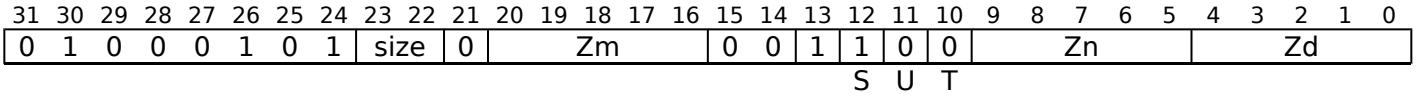
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SABDLB

Signed absolute difference long (bottom)

Compute the absolute difference between even-numbered signed integer values in elements of the second source vector and corresponding elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



SABDLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2*e + 0, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    integer res = Abs(element1 - element2);
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

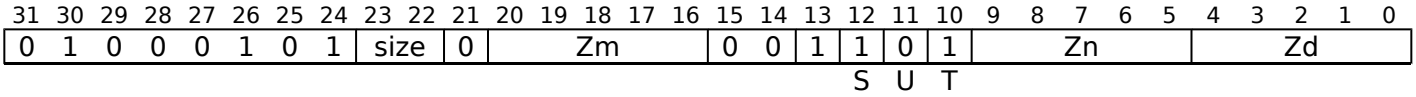
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SABDLT

Signed absolute difference long (top)

Compute the absolute difference between odd-numbered signed integer values in elements of the second source vector and corresponding elements of the first source vector, and place the results in overlapping double-width elements of the destination vector. This instruction is unpredicated.



SABDLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2*e + 1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    integer res = Abs(element1 - element2);
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

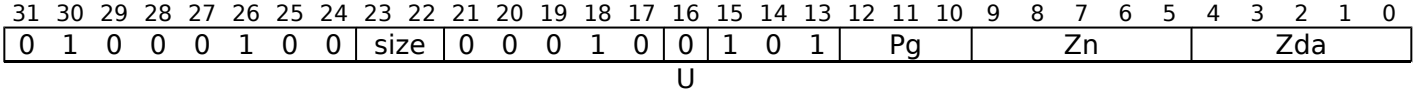
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SADALP

Signed add and accumulate long pairwise

Add pairs of adjacent signed integer values and accumulate the results into the overlapping double-width elements of the destination vector.



SADALP <Zda>.<T>, <Pg>/M, <Zn>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the second source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand_acc = Z[da, VL];
bits(VL) operand_src = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '0' then
        Elem[result, e, esize] = Elem[operand_acc, e, esize];
    else
        integer element1 = SInt(Elem[operand_src, 2*e + 0, esize DIV 2]);
        integer element2 = SInt(Elem[operand_src, 2*e + 1, esize DIV 2]);
        bits(esize) sum = (element1 + element2)<esize-1:0>;
        Elem[result, e, esize] = Elem[operand_acc, e, esize] + sum;

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

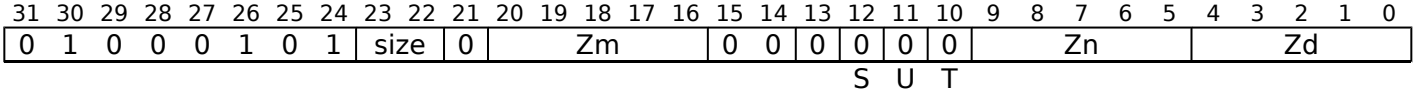
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SADDLB

Signed add long (bottom)

Add the corresponding even-numbered signed elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



SADDLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 0;
integer sel2 = 0;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 + element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

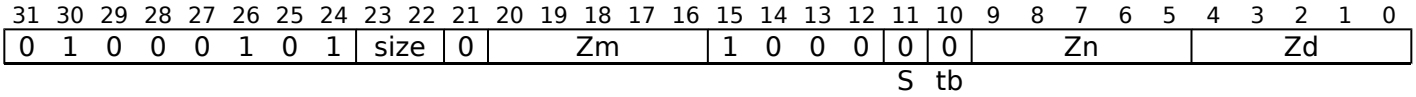
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SADDLBT

Signed add long (bottom + top)

Add the even-numbered signed elements of the first source vector to the odd-numbered signed elements of the second source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



SADDLBT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 0;
integer sel2 = 1;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 + element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

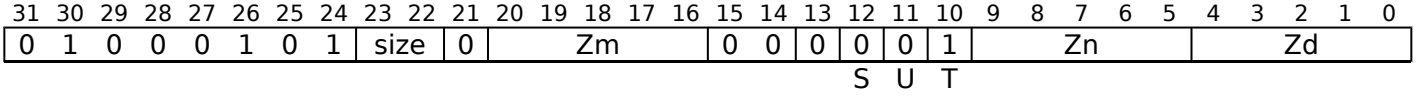
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SADDLT

Signed add long (top)

Add the corresponding odd-numbered signed elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



SADDLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 1;
integer sel2 = 1;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 + element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

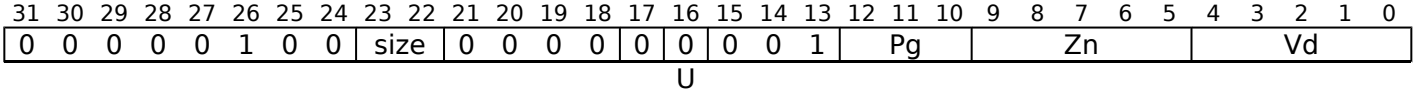
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SADDV

Signed add reduction to scalar

Signed add horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Narrow elements are first sign-extended to 64 bits. Inactive elements in the source vector are treated as zero.



SADDV <Dd>, <Pg>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
```

Assembler Symbols

<Dd> Is the 64-bit name of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	RESERVED

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = Z[n, VL];
integer sum = 0;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element = SInt(Elem[operand, e, esize]);
        sum = sum + element;

V[d, 64] = sum<63:0>;
```

Operational information

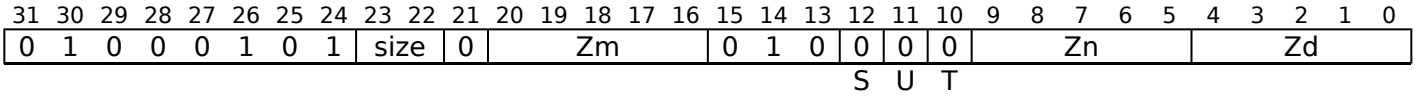
If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

SADDWB

Signed add wide (bottom)

Add the even-numbered signed elements of the second source vector to the overlapping double-width elements of the first source vector and place the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.



SADDWB <Zd>.<T>, <Zn>.<T>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    Elem[result, e, esize] = (element1 + element2)<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

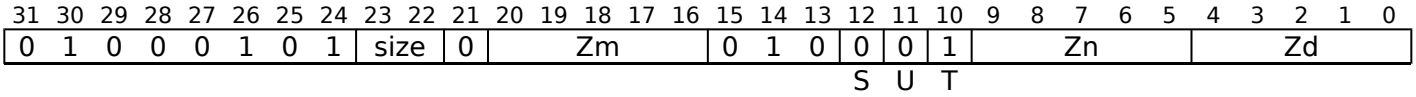
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SADDWT

Signed add wide (top)

Add the odd-numbered signed elements of the second source vector to the overlapping double-width elements of the first source vector and place the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.



SADDWT <Zd>.<T>, <Zn>.<T>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    Elem[result, e, esize] = (element1 + element2)<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

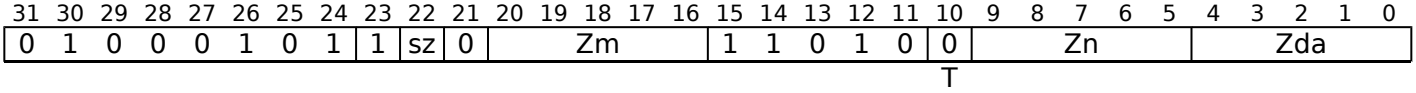
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SBCLB

Subtract with carry long (bottom)

Subtract the even-numbered elements of the first source vector and the inverted 1-bit carry from the least-significant bit of the odd-numbered elements of the second source vector from the even-numbered elements of the destination and accumulator vector. The 1-bit carry output is placed in the corresponding odd-numbered element of the destination vector.



SBCLB <Zda>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer pairs = VL DIV (esize * 2);
bits(VL) operand = Z[n, VL];
bits(VL) carries = Z[m, VL];
bits(VL) result = Z[da, VL];

for p = 0 to pairs-1
    bits(esize) element1 = Elem[result, 2*p + 0, esize];
    bits(esize) element2 = Elem[operand, 2*p + 0, esize];
    bit carry_in = Elem[carries, 2*p + 1, esize]<0>;

    (res, nzcv) = AddWithCarry(element1, NOT(element2), carry_in);
    carry_out = nzcv<1>;

    Elem[result, 2*p + 0, esize] = res;
    Elem[result, 2*p + 1, esize] = ZeroExtend(carry_out, esize);

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

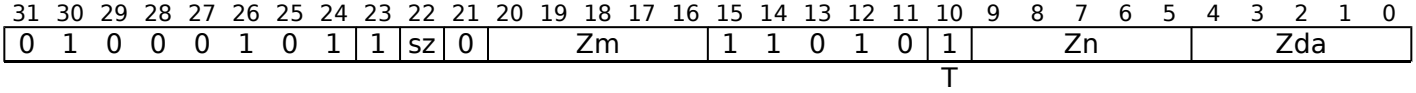
- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SBCLT

Subtract with carry long (top)

Subtract the odd-numbered elements of the first source vector and the inverted 1-bit carry from the least-significant bit of the odd-numbered elements of the second source vector from the even-numbered elements of the destination and accumulator vector. The 1-bit carry output is placed in the corresponding odd-numbered element of the destination vector.



SBCLT <Zda>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32 << UInt(sz);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer pairs = VL DIV (esize * 2);
bits(VL) operand = Z[n, VL];
bits(VL) carries = Z[m, VL];
bits(VL) result = Z[da, VL];

for p = 0 to pairs-1
    bits(esize) element1 = Elem[result, 2*p + 0, esize];
    bits(esize) element2 = Elem[operand, 2*p + 1, esize];
    bit carry_in = Elem[carries, 2*p + 1, esize]<0>;

    (res, nzcv) = AddWithCarry(element1, NOT(element2), carry_in);
    carry_out = nzcv<1>;

    Elem[result, 2*p + 0, esize] = res;
    Elem[result, 2*p + 1, esize] = ZeroExtend(carry_out, esize);

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

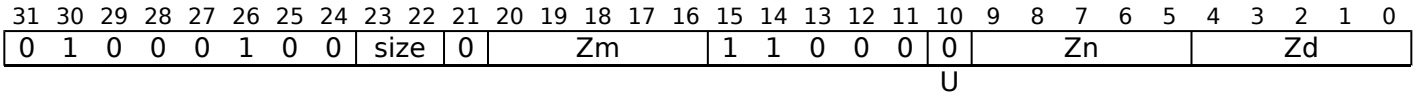
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SCLAMP

Signed clamp to minimum/maximum vector

Clamp each signed element in the destination vector to between the signed minimum value in the corresponding element of the first source vector and the signed maximum value in the corresponding element of the second source vector and destructively write the results in the corresponding elements of the destination vector. This instruction is unpredicated.

SVE2
(FEAT_SME)



SCLAMP <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[d, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    integer element3 = SInt(Elem[operand3, e, esize]);
    integer res = Min(Max(element1 , element3), element2);
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

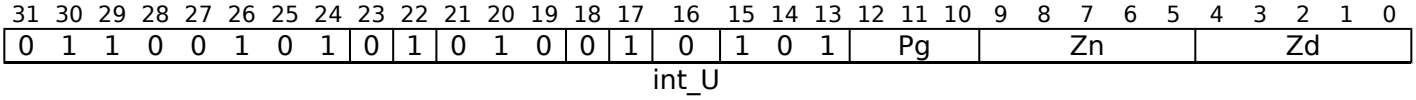
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SCVTF

Signed integer convert to floating-point (predicated)

Convert to floating-point from the signed integer in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.
If the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are zero-extended to fill each destination element.
It has encodings from 7 classes: 16-bit to half-precision , 32-bit to half-precision , 32-bit to single-precision , 32-bit to double-precision , 64-bit to half-precision , 64-bit to single-precision and 64-bit to double-precision

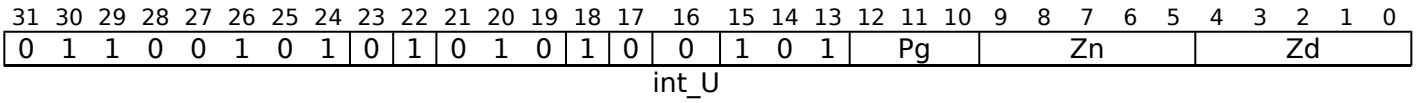
16-bit to half-precision



SCVTF <Zd>.H, <Pg>/M, <Zn>.H

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 16;
constant integer d_esize = 16;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

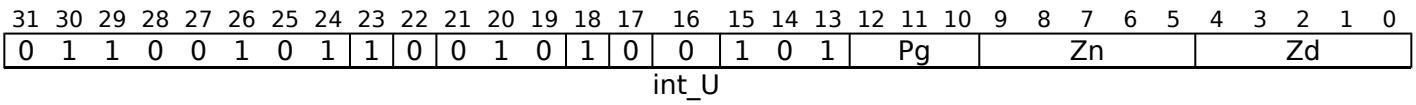
32-bit to half-precision



SCVTF <Zd>.H, <Pg>/M, <Zn>.S

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 32;
constant integer d_esize = 16;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

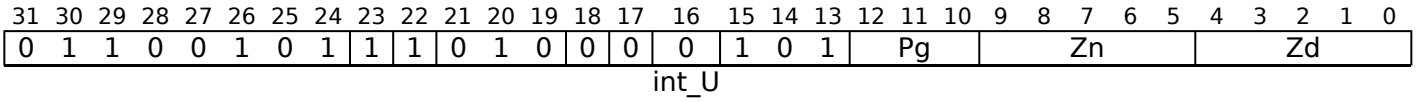
32-bit to single-precision



SCVTF <Zd>.S, <Pg>/M, <Zn>.S

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 32;
constant integer d_esize = 32;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

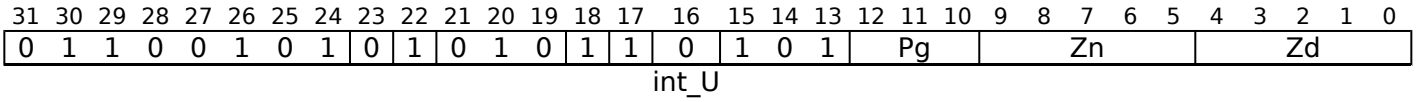
32-bit to double-precision



SCVTF <Zd>.D, <Pg>/M, <Zn>.S

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 32;
constant integer d_esize = 64;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

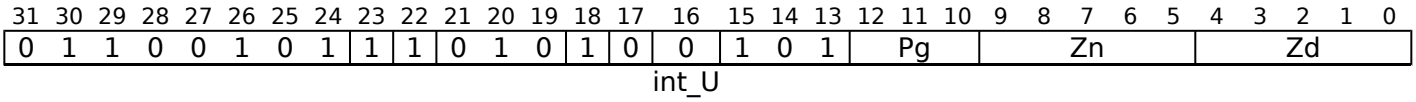
64-bit to half-precision



SCVTF <Zd>.H, <Pg>/M, <Zn>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 64;
constant integer d_esize = 16;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

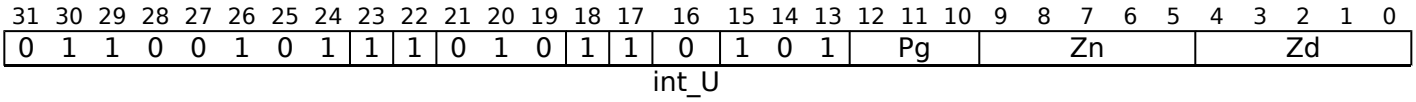
64-bit to single-precision



SCVTF <Zd>.S, <Pg>/M, <Zn>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 64;
constant integer d_esize = 32;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

64-bit to double-precision



SCVTF <Zd>.D, <Pg>/M, <Zn>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 64;
constant integer d_esize = 64;
boolean unsigned = FALSE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        bits(d_esize) fpval = FixedToFP(element<s_esize-1:0>, 0, unsigned, FPCR[], rounding, d_esize);
        Elem[result, e, esize] = ZeroExtend(fpval, esize);

Z[d, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.

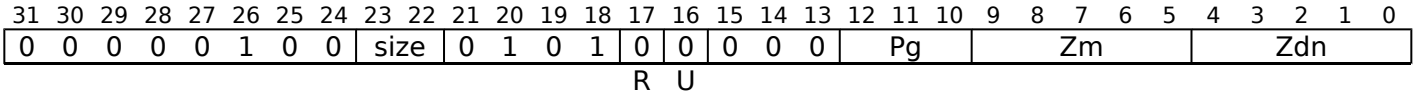
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SDIV

Signed divide (predicated)

Signed divide active elements of the first source vector by corresponding elements of the second source vector and destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



SDIV <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size<0>”:

size<0>	<T>
0	S
1	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    if ElemP[mask, e, esize] == '1' then
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        integer quotient;
        if element2 == 0 then
            quotient = 0;
        else
            quotient = RoundTowardsZero(Real(element1) / Real(element2));
        Elem[result, e, esize] = quotient<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

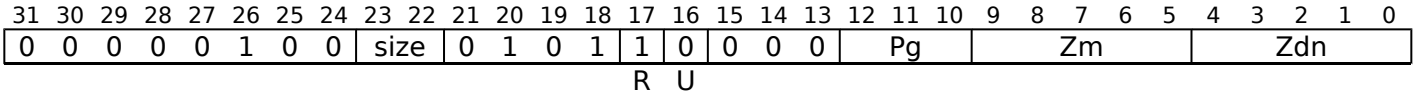
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SDIVR

Signed reversed divide (predicated)

Signed reversed divide active elements of the second source vector by corresponding elements of the first source vector and destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



SDIVR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size<0>”:

size<0>	<T>
0	S
1	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    if ElemP[mask, e, esize] == '1' then
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        integer quotient;
        if element1 == 0 then
            quotient = 0;
        else
            quotient = RoundTowardsZero(Real(element2) / Real(element1));
        Elem[result, e, esize] = quotient<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SDOT (indexed)

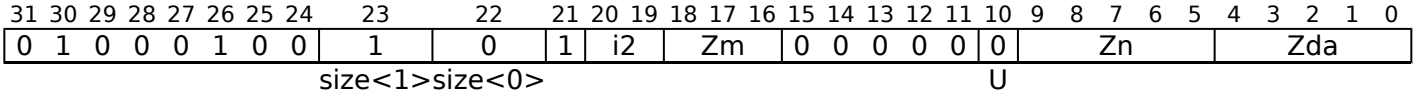
Signed integer indexed dot product

The signed integer indexed dot product instruction computes the dot product of a group of four signed 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the first source vector multiplied by a group of four signed 8-bit or 16-bit integer values in an indexed 32-bit or 64-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit or 64-bit element of the destination vector.

The groups within the second source vector are specified using an immediate index which selects the same group position within each 128-bit vector segment. The index range is from 0 to one less than the number of groups per 128-bit segment, encoded in 1 to 2 bits depending on the size of the group. This instruction is unpredicated.

It has encodings from 2 classes: 32-bit and 64-bit

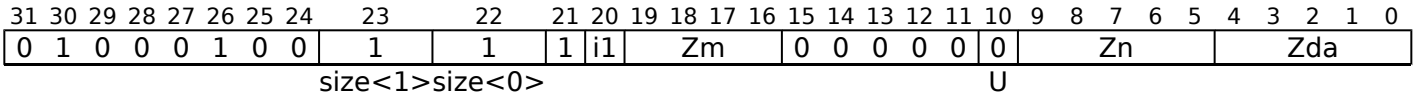
32-bit



SDOT <Zda>.S, <Zn>.B, <Zm>.B[<imm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

64-bit



SDOT <Zda>.D, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field. For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the 32-bit variant: is the immediate index of a 32-bit group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field. For the 64-bit variant: is the immediate index of a 64-bit group of four 16-bit values within each 128-bit vector segment, in the range 0 to 1, encoded in the "i1" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 3
        integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        integer element2 = SInt(Elem[operand2, 4 * s + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = res;

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

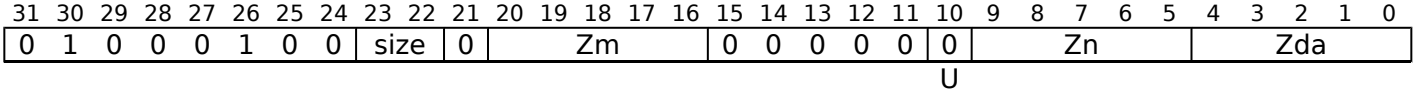
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SDOT (vectors)

Signed integer dot product

The signed integer dot product instruction computes the dot product of a group of four signed 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the first source vector multiplied by a group of four signed 8-bit or 16-bit integer values in the corresponding 32-bit or 64-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit or 64-bit element of the destination vector.

This instruction is unpredicated.



SDOT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size<0>”:

size<0>	<T>
0	S
1	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size<0>”:

size<0>	<Tb>
0	B
1	H

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 3
        integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = res;

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

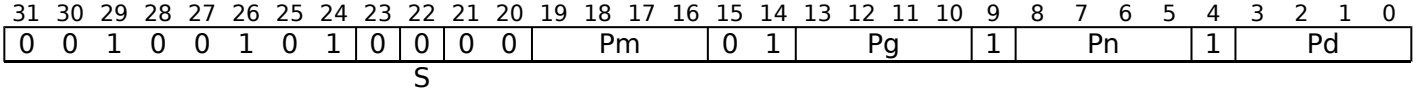
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SEL (predicates)

Conditionally select elements from two predicates

Read active elements from the first source predicate and inactive elements from the second source predicate and place in the corresponding elements of the destination predicate. Does not set the condition flags.
This instruction is used by the alias MOV (predicate, predicated, merging).



SEL <Pd>.B, <Pg>, <Pn>.B, <Pm>.B

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer g = UInt(Pg);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
```

Assembler Symbols

<Pd>	Is the name of the destination scalable predicate register, encoded in the "Pd" field.
<Pg>	Is the name of the governing scalable predicate register, encoded in the "Pg" field.
<Pn>	Is the name of the first source scalable predicate register, encoded in the "Pn" field.
<Pm>	Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Alias Conditions

Alias	Is preferred when
MOV (predicate, predicated, merging)	`Pd == Pm`

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;

for e = 0 to elements-1
    bit element1 = ElemP[operand1, e, esize];
    bit element2 = ElemP[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        ElemP[result, e, esize] = element1;
    else
        ElemP[result, e, esize] = element2;

P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

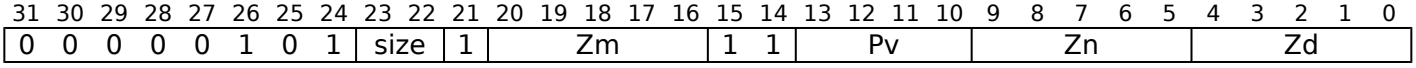
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SEL (vectors)

Conditionally select elements from two vectors

Select elements from the first source vector where the corresponding vector select predicate element is true, and from the second source vector where the predicate element is false, placing them in the corresponding elements of the destination vector.
This instruction is used by the alias MOV (vector, predicated).



SEL <Zd>.<T>, <Pv>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer v = UInt(Pv);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pv> Is the name of the vector select predicate register, encoded in the "Pv" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Alias Conditions

Alias	Is preferred when
MOV (vector, predicated)	`Zd == Zm`

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[v, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) operand2 = if AnyActiveElement(NOT(mask), esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Elem[operand1, e, esize];
    else
        Elem[result, e, esize] = Elem[operand2, e, esize];

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SETFFR

Initialise the first-fault register to all true

Initialise the first-fault register (FFR) to all true prior to a sequence of first-fault or non-fault loads. This instruction is unpredicated.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	0	0	1	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0

SETFFR

```
if !HaveSVE() then UNDEFINED;
```

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
FFR[PL] = Ones(PL);
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

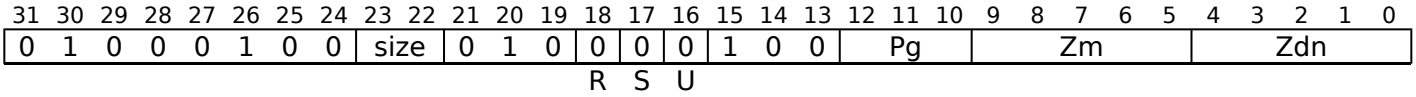
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SHADD

Signed halving addition

Add active signed elements of the first source vector to corresponding signed elements of the second source vector, shift right one bit, and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



SHADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    if ElemP[mask, e, esize] == '1' then
        integer res = (element1 + element2) >> 1;
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

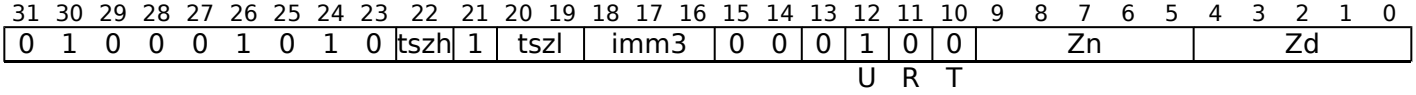
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SHRNB

Shift right narrow by immediate (bottom)

Shift each unsigned integer value in the source vector elements right by an immediate value, and place the truncated results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



SHRNB <Zd>.<T>, <Zn>.<Tb>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '000' UNDEFINED;
    when '001' esize = 8;
    when '01x' esize = 16;
    when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = UInt(element) >> shift;
    Elem[result, 2*e + 0, esize] = res<esize-1:0>;
    Elem[result, 2*e + 1, esize] = Zeros(esize);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

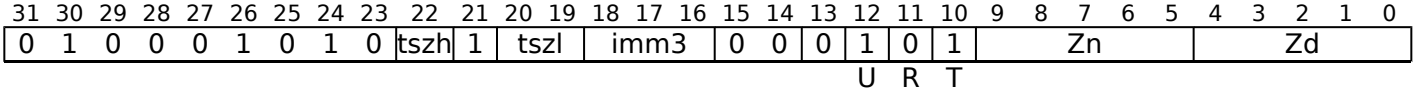
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SHRNT

Shift right narrow by immediate (top)

Shift each unsigned integer value in the source vector elements right by an immediate value, and place the truncated results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



SHRNT <Zd>.<T>, <Zn>.<Tb>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '000' UNDEFINED;
    when '001' esize = 8;
    when '01x' esize = 16;
    when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = UInt(element) >> shift;
    Elem[result, 2*e + 1, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

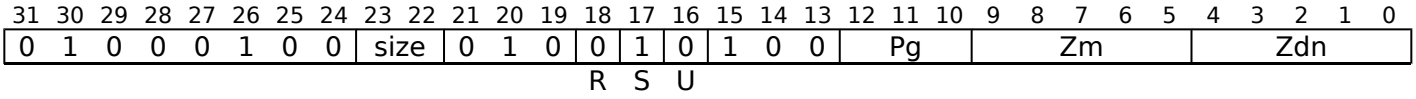
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SHSUB

Signed halving subtract

Subtract active signed elements of the second source vector from corresponding signed elements of the first source vector, shift right one bit, and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



SHSUB <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    if ElemP[mask, e, esize] == '1' then
        integer res = (element1 - element2) >> 1;
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

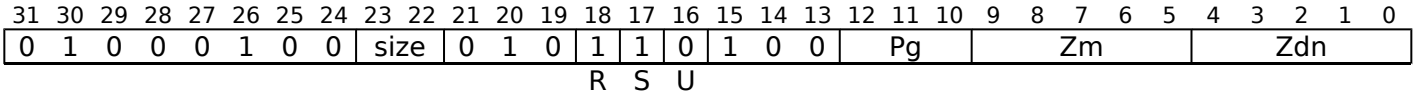
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SHSUBR

Signed halving subtract reversed vectors

Subtract active signed elements of the first source vector from corresponding signed elements of the second source vector, shift right one bit, and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



SHSUBR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    if ElemP[mask, e, esize] == '1' then
        integer res = (element2 - element1) >> 1;
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

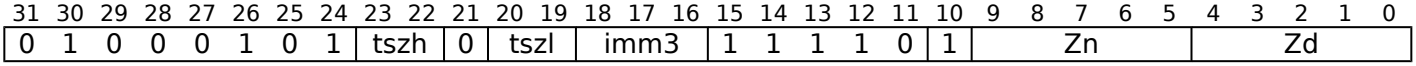
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SLI

Shift left and insert (immediate)

Shift each source vector element left by an immediate value, and insert the result into the corresponding vector element in the destination vector register, merging the shifted bits from each source element with existing bits in each destination vector element. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. This instruction is unpredicated.



SLI <Zd>.<T>, <Zn>.<T>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
case tsize of
    when '0000' UNDEFINED;
    when '0001' esize = 8;
    when '001x' esize = 16;
    when '01xx' esize = 32;
    when '1xxx' esize = 64;
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer shift = UInt(tsize:imm3) - esize;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand = Z[n, VL];
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    bits(esize) element1 = Elem[result, e, esize];
    bits(esize) element2 = Elem[operand, e, esize];
    bits(esize) mask = LSL(Ones(esize), shift);
    bits(esize) shiftedval = LSL(element2, shift);
    Elem[result, e, esize] = (element1 AND (NOT mask)) OR shiftedval;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SM4E

SM4 encryption and decryption

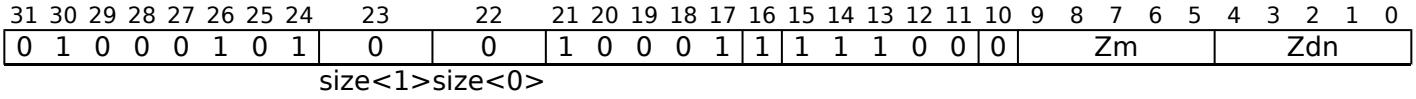
The SM4E instruction reads 16 bytes of input data from each 128-bit segment of the first source vector, together with four iterations of 32-bit round keys from the corresponding 128-bit segments of the second source vector. Each block of data is encrypted by four rounds in accordance with the SM4 standard, and destructively placed in the corresponding segments of the first source vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.SM4 indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE2

(FEAT_SVE_SM4)



SM4E <Zdn>.S, <Zdn>.S, <Zm>.S

```
if !HaveSVE() || !HaveSVE2SM4() then UNDEFINED;
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn>	Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer segments = VL DIV 128;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for s = 0 to segments-1
    bits(128) key = Elem[operand2, s, 128];
    bits(32) intval;
    bits(8) sboxout;
    bits(128) roundresult = Elem[operand1, s, 128];
    bits(32) roundkey;

    for index = 0 to 3
        roundkey = Elem[key, index, 32];
        intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR roundkey;

        for i = 0 to 3
            Elem[intval, i,8]  = Sbox(Elem[intval,i,8]);

        intval = intval EOR ROL(intval, 2) EOR ROL(intval,10) EOR ROL(intval,18) EOR ROL(intval, 24);
        intval = intval EOR roundresult<31:0>;

        roundresult<31:0> = roundresult<63:32>;
        roundresult<63:32> = roundresult<95:64>;
        roundresult<95:64> = roundresult<127:96>;
        roundresult<127:96> = intval;

    Elem[result, s, 128] = roundresult;

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SM4EKEY

SM4 key updates

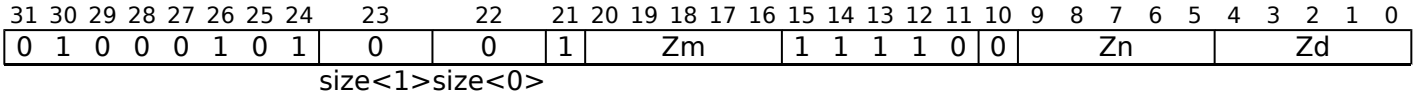
The SM4EKEY instruction reads four rounds of 32-bit input key values from each 128-bit segment of the first source vector, along with four rounds of 32-bit constants from the corresponding 128-bit segment of the second source vector. The four rounds of output key values are derived in accordance with the SM4 standard, and placed in the corresponding segments of the destination vector. This instruction is unpredicated.

ID_AA64ZFR0_EL1.SM4 indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE2

(FEAT_SVE_SM4)



SM4EKEY <Zd>.S, <Zn>.S, <Zm>.S

```
if !HaveSVE() || !HaveSVE2SM4() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer segments = VL DIV 128;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for s = 0 to segments-1
    bits(128) source = Elem[operand2, s, 128];
    bits(32) intval;
    bits(8) sboxout;
    bits(32) const;
    bits(128) roundresult = Elem[operand1, s, 128];

    for index = 0 to 3
        const = Elem[source, index, 32];
        intval = roundresult<127:96> EOR roundresult<95:64> EOR roundresult<63:32> EOR const;
        for i = 0 to 3
            Elem[intval, i, 8] = Sbox(Elem[intval, i, 8]);

        intval = intval EOR ROL(intval, 13) EOR ROL(intval, 23);
        intval = intval EOR roundresult<31:0>;

        roundresult<31:0> = roundresult<63:32>;
        roundresult<63:32> = roundresult<95:64>;
        roundresult<95:64> = roundresult<127:96>;
        roundresult<127:96> = intval;

    Elem[result, s, 128] = roundresult;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

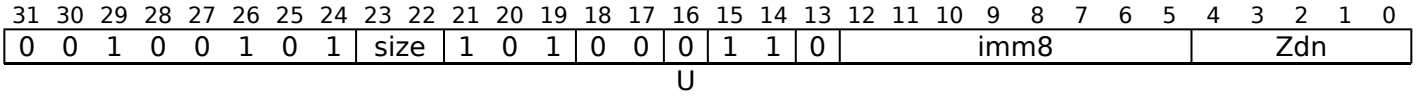
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMAX (immediate)

Signed maximum with immediate (unpredicated)

Determine the signed maximum of an immediate and each element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a signed 8-bit value in the range -128 to +127, inclusive. This instruction is unpredicated.



SMAX <Zdn>.<T>, <Zdn>.<T>, #<imm>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
boolean unsigned = FALSE;
integer imm = Int(imm8, unsigned);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<imm> Is the signed immediate operand, in the range -128 to 127, encoded in the "imm8" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    Elem[result, e, esize] = Max(element1, imm)<esize-1:0>;

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

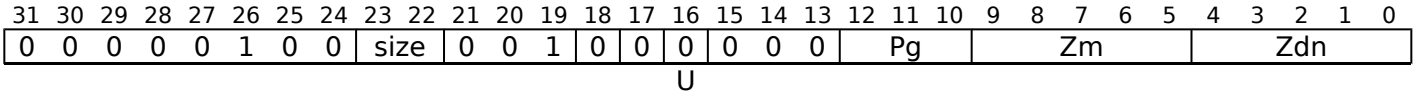
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

SMAX (vectors)

Signed maximum vectors (predicated)

Determine the signed maximum of active elements of the second source vector and corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



SMAX <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    if ElemP[mask, e, esize] == '1' then
        integer maximum = Max(element1, element2);
        Elem[result, e, esize] = maximum<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

        - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
        - The values of the NZCV flags.
    - The response of this instruction to asynchronous exceptions does not vary based on:
        - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
        - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

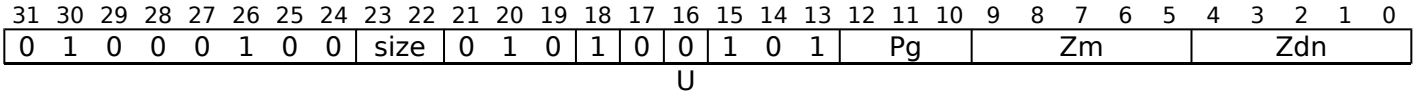
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMAXP

Signed maximum pairwise

Compute the maximum value of each pair of adjacent signed integer elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.



SMAXP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
integer element1;
integer element2;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '0' then
        Elem[result, e, esize] = Elem[operand1, e, esize];
    else
        if IsEven(e) then
            element1 = SInt(Elem[operand1, e + 0, esize]);
            element2 = SInt(Elem[operand1, e + 1, esize]);
        else
            element1 = SInt(Elem[operand2, e - 1, esize]);
            element2 = SInt(Elem[operand2, e + 0, esize]);
        integer res = Max(element1, element2);
        Elem[result, e, esize] = res<esize-1:0>;

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

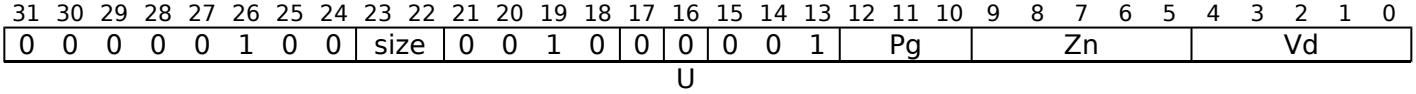
- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMAXV

Signed maximum reduction to scalar

Signed maximum horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the minimum signed integer for the element size.



SMAXV <V><d>, <Pg>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean unsigned = FALSE;
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
integer maximum = if unsigned then 0 else -(2^(esize-1));

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element = Int(Elem[operand, e, esize], unsigned);
        maximum = Max(maximum, element);

V[d, esize] = maximum<esize-1:0>;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

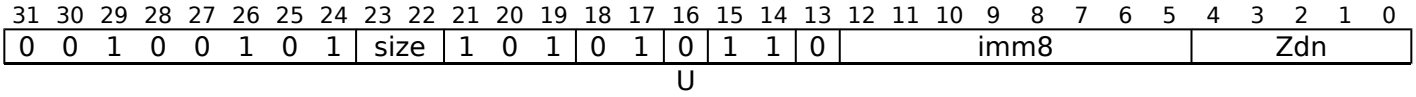
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMIN (immediate)

Signed minimum with immediate (unpredicated)

Determine the signed minimum of an immediate and each element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is a signed 8-bit value in the range -128 to +127, inclusive. This instruction is unpredicated.




```
SMIN <Zdn>.<T>, <Zdn>.<T>, #<imm>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
boolean unsigned = FALSE;
integer imm = Int(imm8, unsigned);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<imm> Is the signed immediate operand, in the range -128 to 127, encoded in the "imm8" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    Elem[result, e, esize] = Min(element1, imm)<esize-1:0>;

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

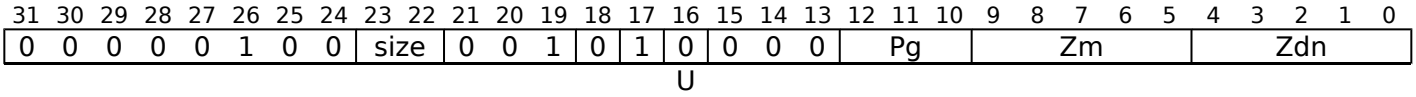
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

SMIN (vectors)

Signed minimum vectors (predicated)

Determine the signed minimum of active elements of the second source vector and corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



SMIN <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    if ElemP[mask, e, esize] == '1' then
        integer minimum = Min(element1, element2);
        Elem[result, e, esize] = minimum<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

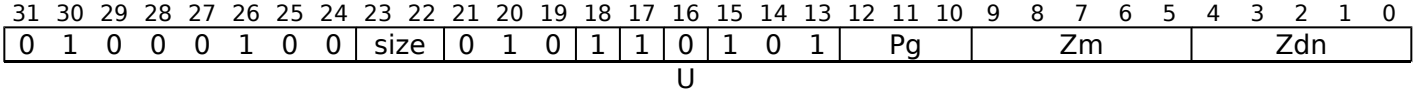
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMINP

Signed minimum pairwise

Compute the minimum value of each pair of adjacent signed integer elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.



SMINP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
integer element1;
integer element2;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '0' then
        Elem[result, e, esize] = Elem[operand1, e, esize];
    else
        if IsEven(e) then
            element1 = SInt(Elem[operand1, e + 0, esize]);
            element2 = SInt(Elem[operand1, e + 1, esize]);
        else
            element1 = SInt(Elem[operand2, e - 1, esize]);
            element2 = SInt(Elem[operand2, e + 0, esize]);
        integer res = Min(element1, element2);
        Elem[result, e, esize] = res<esize-1:0>;

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

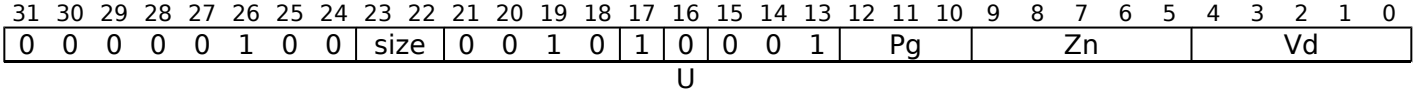
- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMINV

Signed minimum reduction to scalar

Signed minimum horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the maximum signed integer for the element size.



SMINV <V><d>, <Pg>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean unsigned = FALSE;
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
integer minimum = if unsigned then (2^esize - 1) else (2^(esize-1) - 1);

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element = Int(Elem[operand, e, esize], unsigned);
        minimum = Min(minimum, element);

V[d, esize] = minimum<esize-1:0>;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

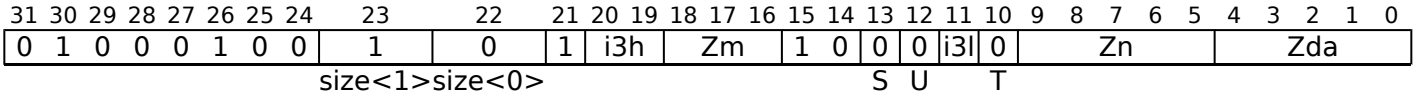
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMLALB (indexed)

Signed multiply-add long to accumulator (bottom, indexed)

Multiply the even-numbered signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment and destructively add to the overlapping double-width elements of the addend vector.
The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.
It has encodings from 2 classes: 32-bit and 64-bit

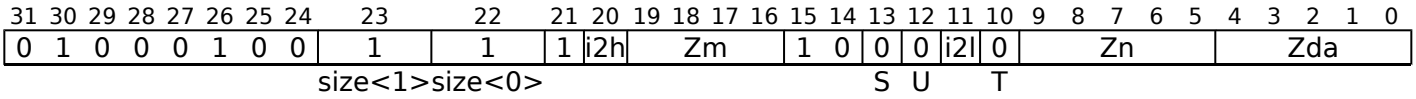
32-bit



SMLALB <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```

64-bit



SMLALB <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field. For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields. For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel,   esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    bits(2*esize) product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + product;

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

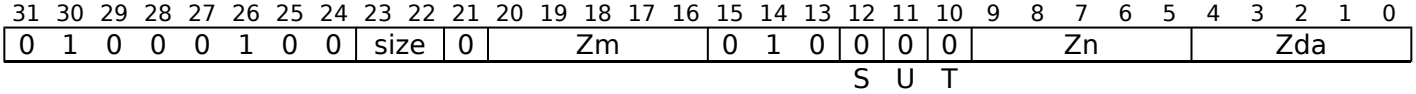
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMLALB (vectors)

Signed multiply-add long to accumulator (bottom)

Multiply the corresponding even-numbered signed elements of the first and second source vectors and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.



SMLALB <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2*e + 0, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    bits(esize) product = (element1 * element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + product;

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

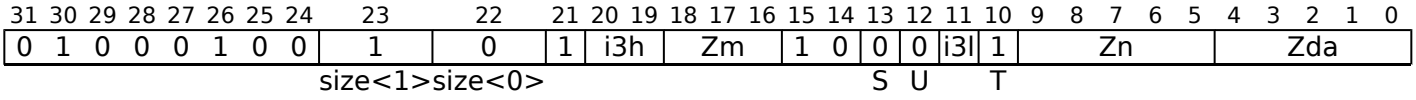
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMLALT (indexed)

Signed multiply-add long to accumulator (top, indexed)

Multiply the odd-numbered signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment and destructively add to the overlapping double-width elements of the addend vector.
The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.
It has encodings from 2 classes: 32-bit and 64-bit

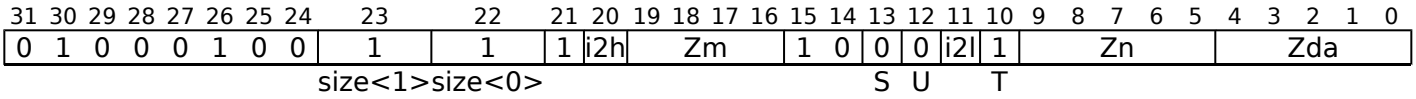
32-bit



SMLALT <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
```

64-bit



SMLALT <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field. For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields. For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel,   esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    bits(2*esize) product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + product;

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

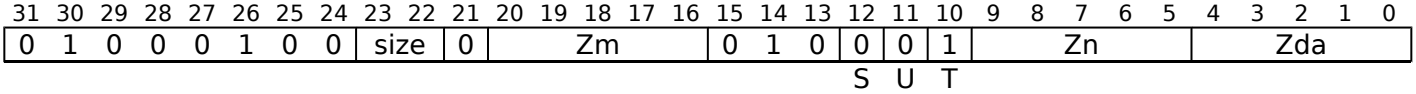
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMLALT (vectors)

Signed multiply-add long to accumulator (top)

Multiply the corresponding odd-numbered signed elements of the first and second source vectors and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.



SMLALT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2*e + 1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    bits(esize) product = (element1 * element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + product;

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

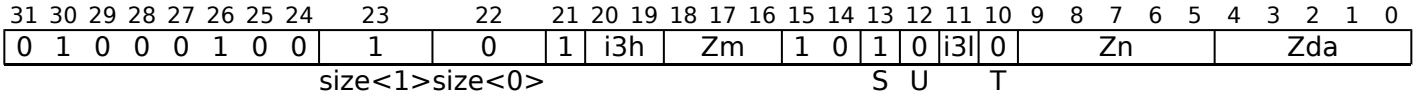
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMLSLB (indexed)

Signed multiply-subtract long from accumulator (bottom, indexed)

Multiply the even-numbered signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment and destructively subtract from the overlapping double-width elements of the addend vector.
The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.
It has encodings from 2 classes: 32-bit and 64-bit

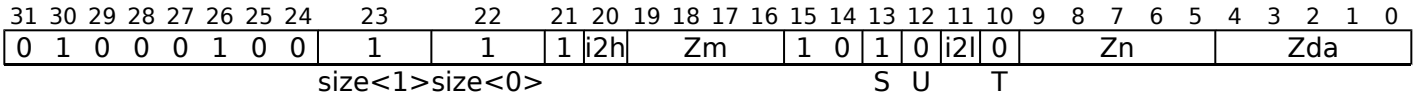
32-bit



SMLSLB <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```

64-bit



SMLSLB <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field. For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields. For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel,   esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    bits(2*esize) product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] - product;

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

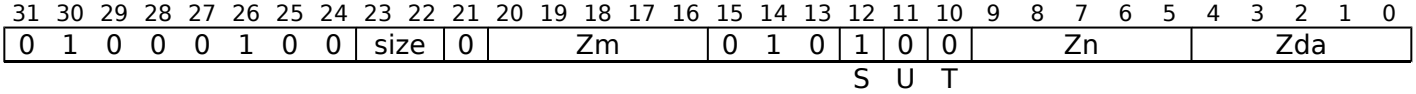
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMLSLB (vectors)

Signed multiply-subtract long from accumulator (bottom)

Multiply the corresponding even-numbered signed elements of the first and second source vectors and destructively subtract from the overlapping double-width elements of the addend vector. This instruction is unpredicated.



SMLSLB <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2*e + 0, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    bits(esize) product = (element1 * element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] - product;

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

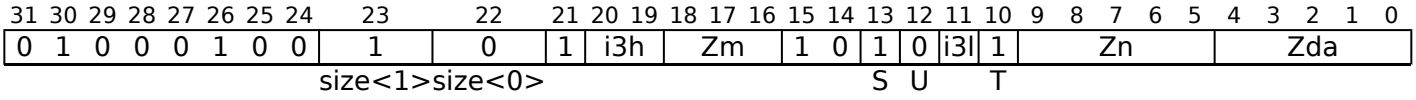
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMLSLT (indexed)

Signed multiply-subtract long from accumulator (top, indexed)

Multiply the odd-numbered signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment and destructively subtract from the overlapping double-width elements of the addend vector.
The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.
It has encodings from 2 classes: 32-bit and 64-bit

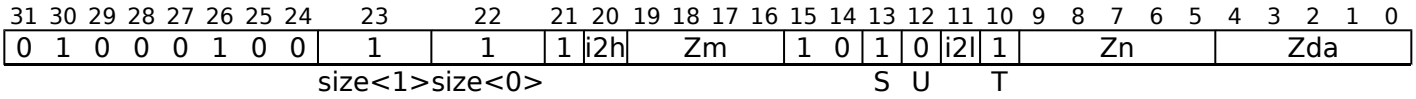
32-bit



SMLSLT <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
```

64-bit



SMLSLT <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field. For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields. For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel,   esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    bits(2*esize) product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] - product;

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

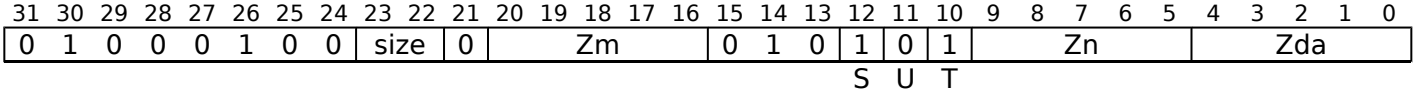
- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMLSLT (vectors)

Signed multiply-subtract long from accumulator (top)

Multiply the corresponding odd-numbered signed elements of the first and second source vectors and destructively subtract from the overlapping double-width elements of the addend vector. This instruction is unpredicated.



SMLSLT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2*e + 1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    bits(esize) product = (element1 * element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] - product;

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMMLA

Signed integer matrix multiply-accumulate

The signed integer matrix multiply-accumulate instruction multiplies the 2×8 matrix of signed 8-bit integer values held in each 128-bit segment of the first source vector by the 8×2 matrix of signed 8-bit integer values in the corresponding segment of the second source vector. The resulting 2×2 widened 32-bit integer matrix product is then destructively added to the 32-bit integer matrix accumulator held in the corresponding segment of the addend and destination vector. This is equivalent to performing an 8-way dot product per destination element.

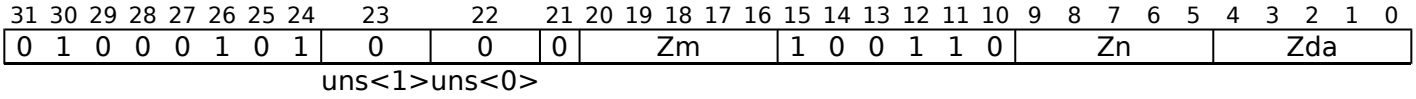
This instruction is unpredicated.

ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE

(FEAT_I8MM)



SMMLA <Zda>.S, <Zn>.B, <Zm>.B

```
if !HaveSVE() || !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_unsigned = FALSE;
boolean op2_unsigned = FALSE;
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer segments = VL DIV 128;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result = Zeros(VL);
bits(128) op1, op2;
bits(128) res, addend;

for s = 0 to segments-1
    op1    = Elem[operand1, s, 128];
    op2    = Elem[operand2, s, 128];
    addend = Elem[operand3, s, 128];
    res    = MatMulAdd(addend, op1, op2, op1_unsigned, op2_unsigned);
    Elem[result, s, 128] = res;

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

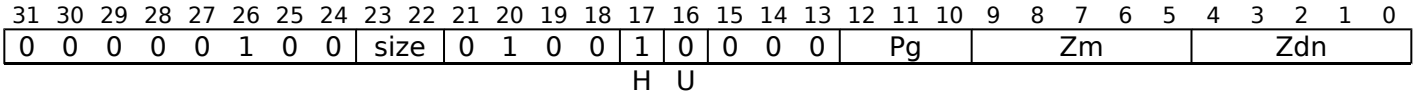
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMULH (predicated)

Signed multiply returning high half (predicated)

Widening multiply signed integer values in active elements of the first source vector by corresponding elements of the second source vector and destructively place the high half of the result in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



SMULH <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    if ElemP[mask, e, esize] == '1' then
        integer product = (element1 * element2) >> esize;
        Elem[result, e, esize] = product<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

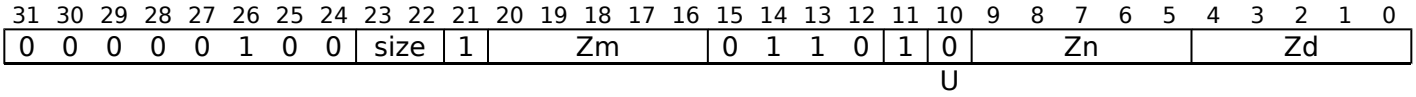
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMULH (unpredicated)

Signed multiply returning high half (unpredicated)

Widening multiply signed integer values of all elements of the first source vector by corresponding elements of the second source vector and place the high half of the result in the corresponding elements of the destination vector. This instruction is unpredicated.



SMULH <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean unsigned = FALSE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    integer product = (element1 * element2) >> esize;
    Elem[result, e, esize] = product<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

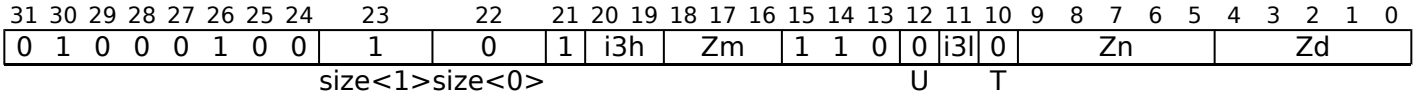
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

SMULLB (indexed)

Signed multiply long (bottom, indexed)

Multiply the even-numbered signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment, and place the results in the overlapping double-width elements of the destination vector register.
The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.
It has encodings from 2 classes: 32-bit and 64-bit

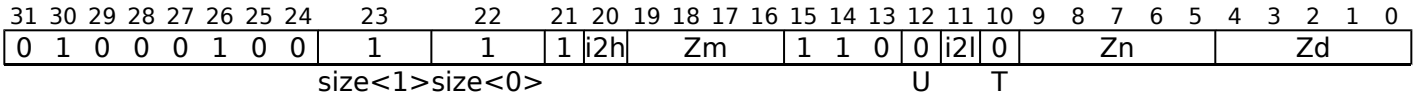
32-bit



SMULLB <Zd>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 0;
```

64-bit



SMULLB <Zd>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 0;
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field. For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields. For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel,   esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    integer res = element1 * element2;
    Elem[result, e, 2*esize] = res<2*esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

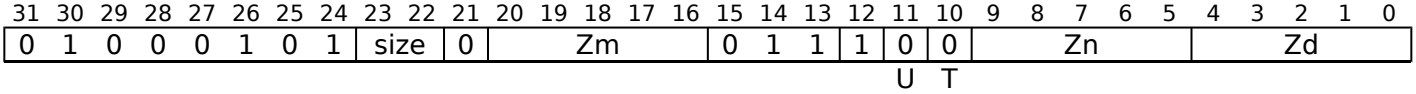
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMULLB (vectors)

Signed multiply long (bottom)

Multiply the corresponding even-numbered signed elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



SMULLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2*e + 0, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    integer res = element1 * element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

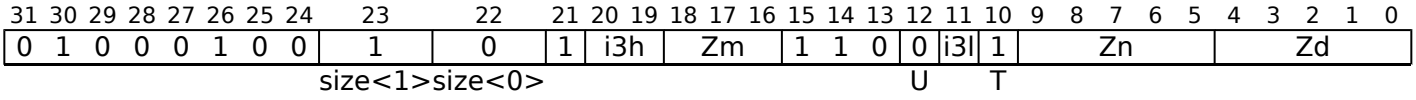
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMULLT (indexed)

Signed multiply long (top, indexed)

Multiply the odd-numbered signed elements within each 128-bit segment of the first source vector by the specified element in the corresponding second source vector segment, and place the results in the overlapping double-width elements of the destination vector register.
The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.
It has encodings from 2 classes: 32-bit and 64-bit

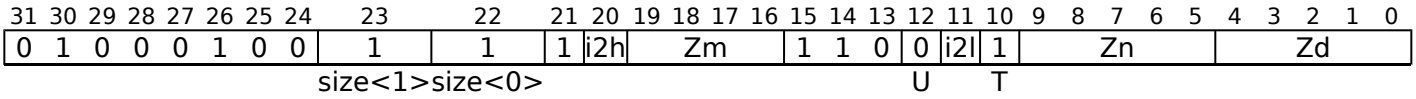
32-bit



SMULLT <Zd>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 1;
```

64-bit



SMULLT <Zd>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 1;
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field. For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields. For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel,   esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    integer res = element1 * element2;
    Elem[result, e, 2*esize] = res<2*esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

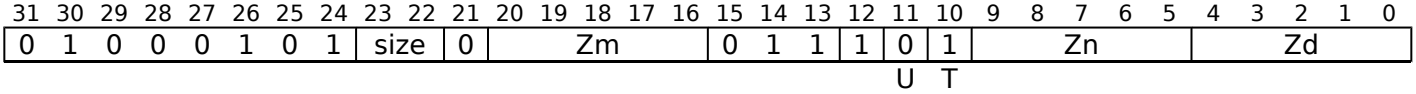
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SMULLT (vectors)

Signed multiply long (top)

Multiply the corresponding odd-numbered signed elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



SMULLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2*e + 1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    integer res = element1 * element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

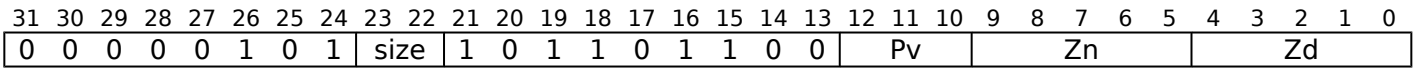
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SPLICE

Splice two vectors under predicate control

Select a region from the first source vector and copy it to the lowest-numbered elements of the result. Then set any remaining elements of the result to a copy of the lowest-numbered elements from the second source vector. The region is selected using the first and last true elements in the vector select predicate register. The result is placed destructively in the destination and first source vector, or constructively in the destination vector.
The Destructive encoding of this instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE: The MOVPRFX instruction must be unpredicated. The MOVPRFX instruction must specify the same destination register as this instruction. The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.
It has encodings from 2 classes: Constructive and Destructive

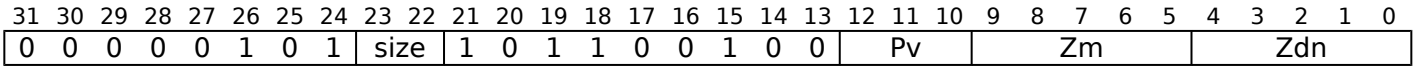
Constructive



SPLICE <Zd>.<T>, <Pv>, { <Zn1>.<T>, <Zn2>.<T> }

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer v = UInt(Pv);
integer dst = UInt(Zd);
integer s1 = UInt(Zn);
integer s2 = (s1 + 1) MOD 32;
```

Destructive



SPLICE <Zdn>.<T>, <Pv>, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer v = UInt(Pv);
integer dst = UInt(Zdn);
integer s1 = dst;
integer s2 = UInt(Zm);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pv> Is the name of the vector select predicate register P0-P7, encoded in the "Pv" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded in the "Zn" field.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

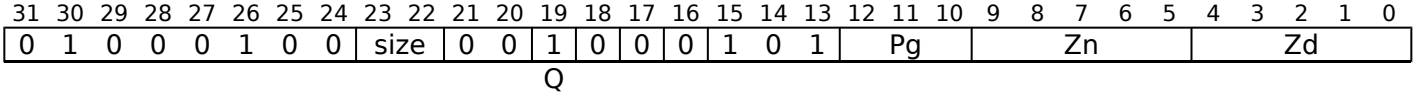
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[v, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[s1, VL] else Zeros(VL);
bits(VL) operand2 = Z[s2, VL];
bits(VL) result;
integer x = 0;
boolean active = FALSE;
constant integer lastnum = LastActiveElement(mask, esize);

if lastnum >= 0 then
    for e = 0 to lastnum
        active = active || ElemP[mask, e, esize] == '1';
        if active then
            Elem[result, x, esize] = Elem[operand1, e, esize];
            x = x + 1;

constant integer nelements = (elements - x) - 1;
for e = 0 to nelements
    Elem[result, x, esize] = Elem[operand2, e, esize];
    x = x + 1;

Z[dst, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQABS

Signed saturating absolute value

Compute the absolute value of the signed integer in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Inactive elements in the destination vector register remain unmodified.



SQABS <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element = SInt(Elem[operand, e, esize]);
        element = Abs(element);
        Elem[result, e, esize] = SignedSat(element, esize);

Z[d, VL] = result;
```

Operational information

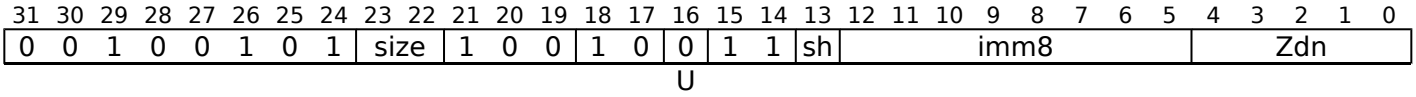
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

SQADD (immediate)

Signed saturating add immediate (unpredicated)

Signed saturating add of an unsigned immediate to each element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.
The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.
The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<uimm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".



SQADD <Zdn>.<T>, <Zdn>.<T>, #<imm>{, <shift>}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = UInt(imm8);
if sh == '1' then imm = imm << 8;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<imm> Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #8

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + imm, esize, unsigned);

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

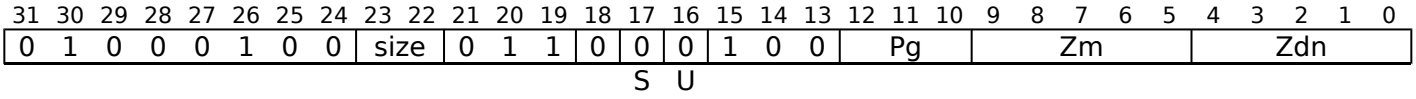
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQADD (vectors, predicated)

Signed saturating addition (predicated)

Add active signed elements of the first source vector to corresponding signed elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Inactive elements in the destination vector register remain unmodified.



SQADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    if ElemP[mask, e, esize] == '1' then
        integer res = SInt(Sat(element1 + element2, esize, unsigned));
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

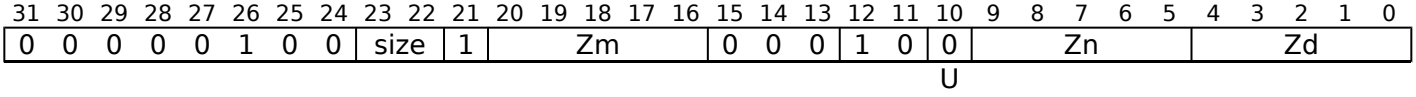
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQADD (vectors, unpredicated)

Signed saturating add vectors (unpredicated)

Signed saturating add all elements of the second source vector to corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.



SQADD <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean unsigned = FALSE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + element2, esize, unsigned);

Z[d, VL] = result;
```

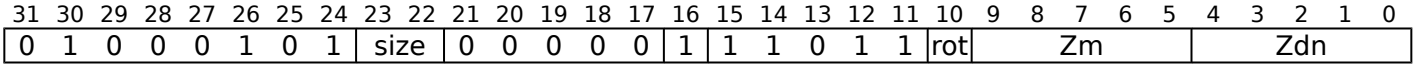
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQCADD

Saturating complex integer add with rotate

Add the real and imaginary components of the integral complex numbers from the first source vector to the complex numbers from the second source vector which have first been rotated by 90 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation, equivalent to multiplying the complex numbers in the second source vector by ±J beforehand. Destructively place the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.



SQCADD <Zdn>.<T>, <Zdn>.<T>, <Zm>.<T>, <const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
boolean sub_i = (rot == '0');
boolean sub_r = (rot == '1');
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<const> Is the const specifier, encoded in “rot”:

rot	<const>
0	#90
1	#270

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (2 * esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for p = 0 to pairs-1
    integer acc_r  = SInt(Elem[operand1, 2 * p + 0, esize]);
    integer acc_i  = SInt(Elem[operand1, 2 * p + 1, esize]);
    integer elt2_r = SInt(Elem[operand2, 2 * p + 0, esize]);
    integer elt2_i = SInt(Elem[operand2, 2 * p + 1, esize]);
    if sub_i then
        acc_r = acc_r - elt2_i;
        acc_i = acc_i + elt2_r;
    if sub_r then
        acc_r = acc_r + elt2_i;
        acc_i = acc_i - elt2_r;

    Elem[result, 2 * p + 0, esize] = SignedSat(acc_r, esize);
    Elem[result, 2 * p + 1, esize] = SignedSat(acc_i, esize);

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQDECB

Signed saturating decrement scalar by multiple of 8-bit predicate constraint element count

Determines the number of active 8-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

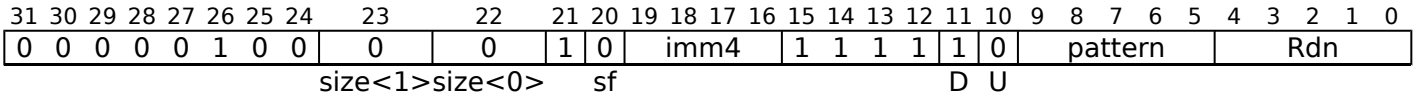
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

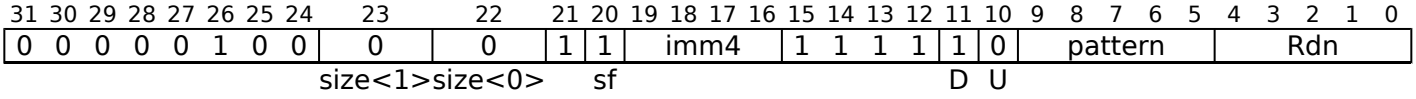
32-bit



SQDECB <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;
```

64-bit



SQDECB <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;
```

Assembler Symbols

<Xdn>	Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Wdn>	Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<pattern>	Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQDECD (scalar)

Signed saturating decrement scalar by multiple of 64-bit predicate constraint element count

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

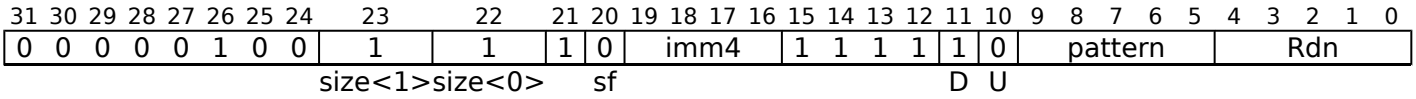
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

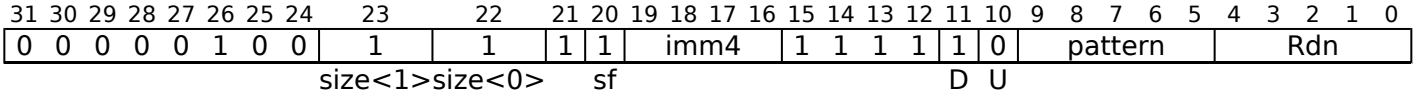
32-bit



SQDECD <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;
```

64-bit



SQDECD <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;
```

Assembler Symbols

<Xdn>	Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Wdn>	Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<pattern>	Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQDECD (vector)

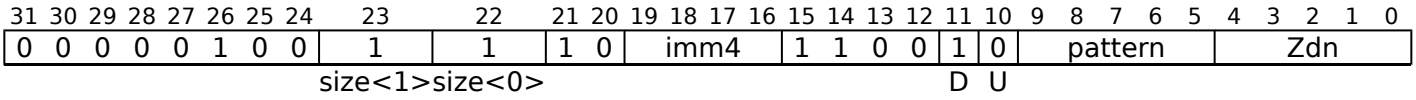
Signed saturating decrement vector by multiple of 64-bit predicate constraint element count

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 64-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.



SQDECD <Zdn>.D{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQDECH (scalar)

Signed saturating decrement scalar by multiple of 16-bit predicate constraint element count

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

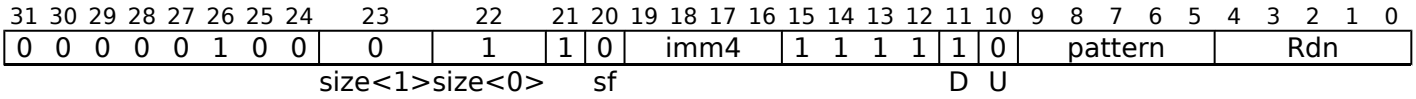
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

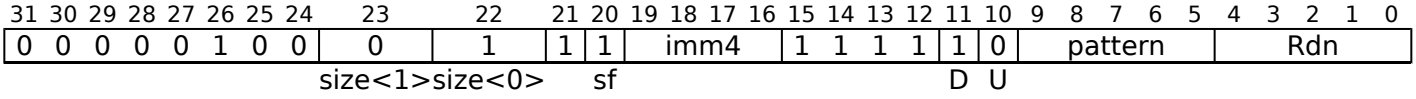
32-bit



SQDECH <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;
```

64-bit



SQDECH <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;
```

Assembler Symbols

<Xdn>	Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Wdn>	Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<pattern>	Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQDECH (vector)

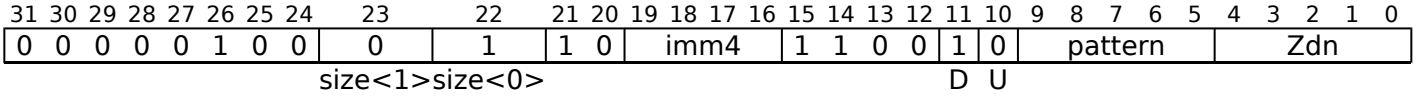
Signed saturating decrement vector by multiple of 16-bit predicate constraint element count

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 16-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.



SQDECH <Zdn>.H{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

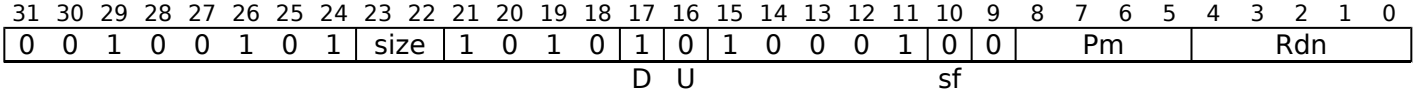
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQDECP (scalar)

Signed saturating decrement scalar by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.
It has encodings from 2 classes: 32-bit and 64-bit

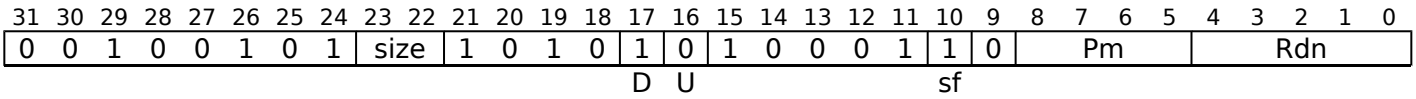
32-bit



SQDECP <Xdn>, <Pm>.<T>, <Wdn>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = FALSE;
integer ssize = 32;
```

64-bit



SQDECP <Xdn>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = FALSE;
integer ssize = 64;
```

Assembler Symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(ssize) operand1 = X[dn, ssize];
bits(PL) operand2 = P[m, PL];
bits(ssize) result;
integer count = 0;

for e = 0 to elements-1
    if ElemP[operand2, e, esize] == '1' then
        count = count + 1;

integer element = Int(operand1, unsigned);
(result, -) = SatQ(element - count, ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
```

Operational information

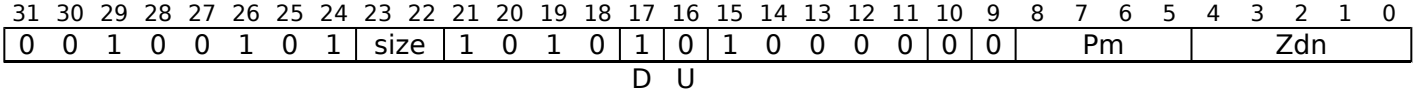
If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQDECP (vector)

Signed saturating decrement vector by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to decrement all destination vector elements. The results are saturated to the element signed integer range.
The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.



SQDECP <Zdn>.<T>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) operand2 = P[m, PL];
bits(VL) result;
integer count = 0;

for e = 0 to elements-1
    if ElemP[operand2, e, esize] == '1' then
        count = count + 1;

for e = 0 to elements-1
    integer element = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element - count, esize, unsigned);

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQDECW (scalar)

Signed saturating decrement scalar by multiple of 32-bit predicate constraint element count

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

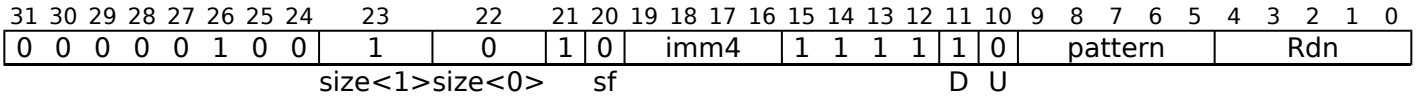
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

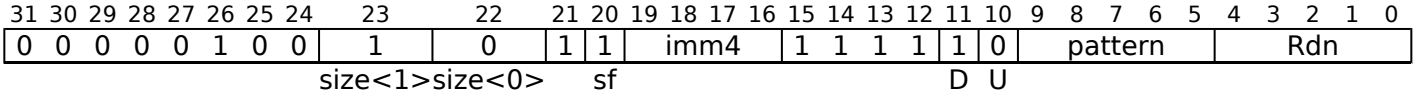
32-bit



SQDECW <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;
```

64-bit



SQDECW <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;
```

Assembler Symbols

<Xdn>	Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Wdn>	Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<pattern>	Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQDECW (vector)

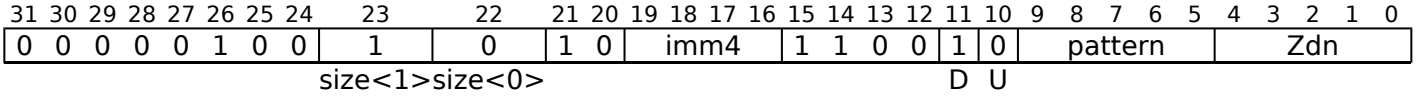
Signed saturating decrement vector by multiple of 32-bit predicate constraint element count

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 32-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.



SQDECW <Zdn>.S{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

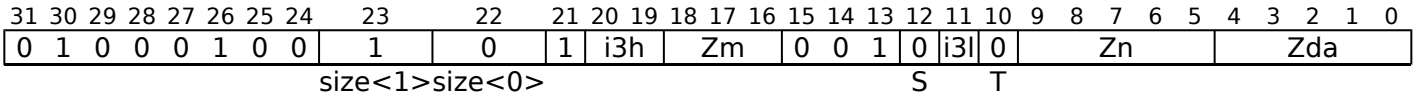
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQDMLALB (indexed)

Signed saturating doubling multiply-add long to accumulator (bottom, indexed)

Multiply then double the even-numbered signed elements within each 128-bit segment of the first source vector and specified signed element in the corresponding second source vector segment. Each intermediate value is saturated to the double-width N-bit value's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Then destructively add to the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$.
The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.
It has encodings from 2 classes: 32-bit and 64-bit

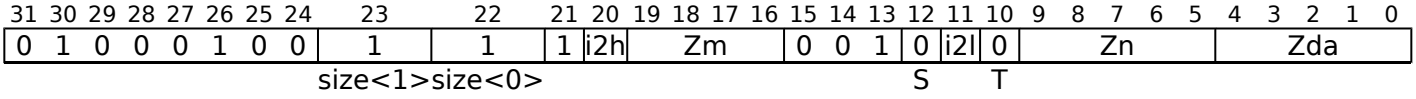
32-bit



SQDMLALB <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```

64-bit



SQDMLALB <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field. For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields. For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel,   esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    integer element3 = SInt(Elem[result, e, 2*esize]);
    integer product = SInt(SignedSat(2 * element1 * element2, 2*esize));
    integer res = element3 + product;
    Elem[result, e, 2*esize] = SignedSat(res, 2*esize);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

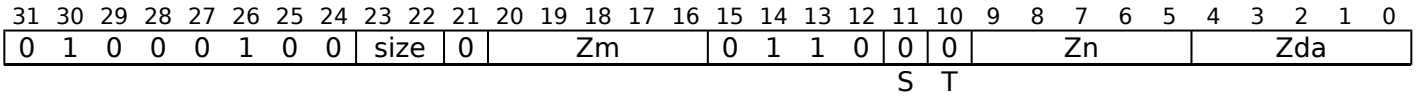
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQDMLALB (vectors)

Signed saturating doubling multiply-add long to accumulator (bottom)

Multiply then double the corresponding even-numbered signed elements of the first and second source vectors. Each intermediate value is saturated to the double-width N-bit value's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Then destructively add to the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.



SQDMLALB <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel1 = 0;
integer sel2 = 0;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2 * e + sel1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2 * e + sel2, esize DIV 2]);
    integer element3 = SInt(Elem[result, e, esize]);
    integer product = SInt(SignedSat(2 * element1 * element2, esize));
    Elem[result, e, esize] = SignedSat(element3 + product, esize);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

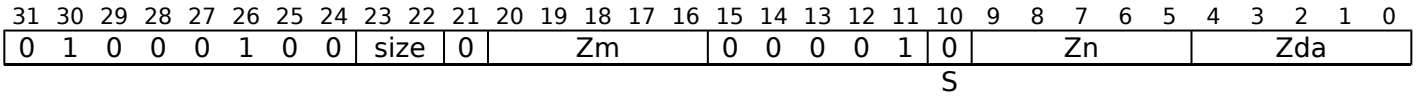
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQDMLALBT

Signed saturating doubling multiply-add long to accumulator (bottom × top)

Multiply then double the corresponding even-numbered signed elements of the first and odd-numbered signed elements of the second source vector. Each intermediate value is saturated to the double-width N-bit value's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Then destructively add to the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.



SQDMLALBT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel1 = 0;
integer sel2 = 1;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2 * e + sel1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2 * e + sel2, esize DIV 2]);
    integer element3 = SInt(Elem[result, e, esize]);
    integer product = SInt(SignedSat(2 * element1 * element2, esize));
    Elem[result, e, esize] = SignedSat(element3 + product, esize);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

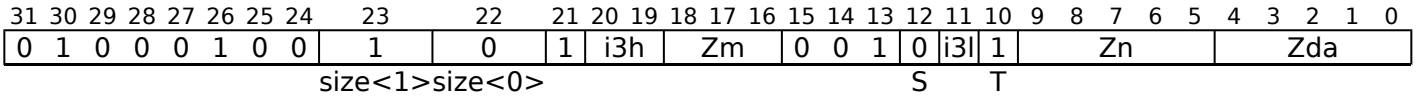
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQDMLALT (indexed)

Signed saturating doubling multiply-add long to accumulator (top, indexed)

Multiply then double the odd-numbered signed elements within each 128-bit segment of the first source vector and the specified signed element in the corresponding second source vector segment. Each intermediate value is saturated to the double-width N-bit value's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Then destructively add to the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$.
The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.
It has encodings from 2 classes: 32-bit and 64-bit

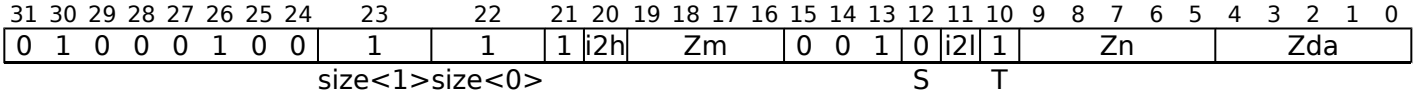
32-bit



SQDMLALT <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
```

64-bit



SQDMLALT <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field. For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields. For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel,   esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    integer element3 = SInt(Elem[result, e, 2*esize]);
    integer product = SInt(SignedSat(2 * element1 * element2, 2*esize));
    integer res = element3 + product;
    Elem[result, e, 2*esize] = SignedSat(res, 2*esize);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

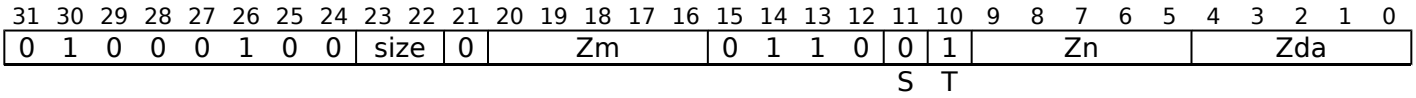
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQDMLALT (vectors)

Signed saturating doubling multiply-add long to accumulator (top)

Multiply then double the corresponding odd-numbered signed elements of the first and second source vectors. Each intermediate value is saturated to the double-width N-bit value's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Then destructively add to the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.



SQDMLALT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel1 = 1;
integer sel2 = 1;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2 * e + sel1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2 * e + sel2, esize DIV 2]);
    integer element3 = SInt(Elem[result, e, esize]);
    integer product = SInt(SignedSat(2 * element1 * element2, esize));
    Elem[result, e, esize] = SignedSat(element3 + product, esize);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

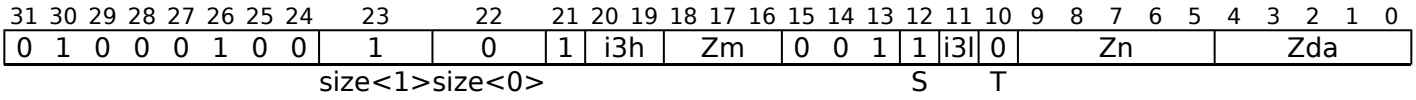
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQDMLSLB (indexed)

Signed saturating doubling multiply-subtract long from accumulator (bottom, indexed)

Multiply then double the even-numbered signed elements within each 128-bit segment of the first source vector and the specified signed element in the corresponding second source vector segment. Each intermediate value is saturated to the double-width N-bit value's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Then destructively subtract from the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$.
The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.
It has encodings from 2 classes: 32-bit and 64-bit

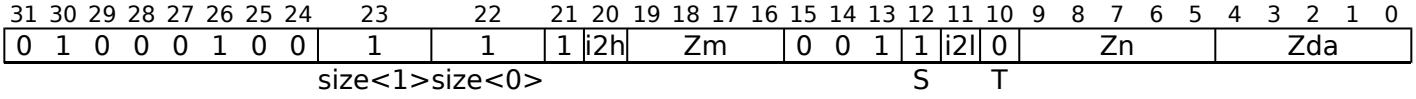
32-bit



SQDMLSLB <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```

64-bit



SQDMLSLB <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field. For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields. For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel,   esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    integer element3 = SInt(Elem[result, e, 2*esize]);
    integer product = SInt(SignedSat(2 * element1 * element2, 2*esize));
    integer res = element3 - product;
    Elem[result, e, 2*esize] = SignedSat(res, 2*esize);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

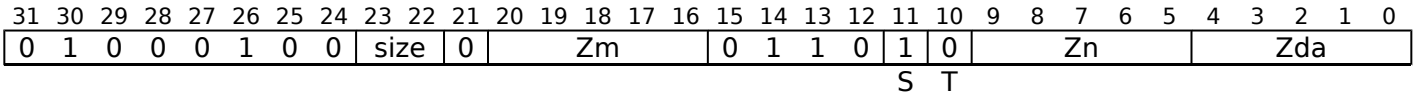
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQDMLSLB (vectors)

Signed saturating doubling multiply-subtract long from accumulator (bottom)

Multiply then double the corresponding even-numbered signed elements of the first and second source vectors. Each intermediate value is saturated to the double-width N-bit value's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Then destructively subtract from the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.



SQDMLSLB <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel1 = 0;
integer sel2 = 0;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2 * e + sel1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2 * e + sel2, esize DIV 2]);
    integer element3 = SInt(Elem[result, e, esize]);
    integer product = SInt(SignedSat(2 * element1 * element2, esize));
    Elem[result, e, esize] = SignedSat(element3 - product, esize);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

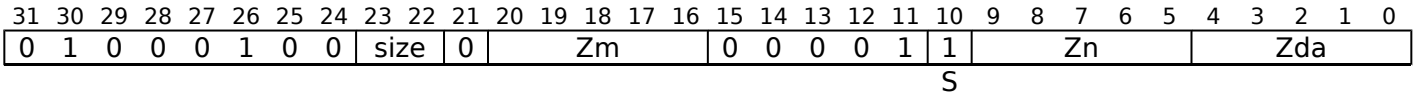
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQDMLSLBT

Signed saturating doubling multiply-subtract long from accumulator (bottom × top)

Multiply then double the corresponding even-numbered signed elements of the first and odd-numbered signed elements of the second source vector. Each intermediate value is saturated to the double-width N-bit value's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Then destructively subtract from the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.



SQDMLSLBT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel1 = 0;
integer sel2 = 1;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2 * e + sel1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2 * e + sel2, esize DIV 2]);
    integer element3 = SInt(Elem[result, e, esize]);
    integer product = SInt(SignedSat(2 * element1 * element2, esize));
    Elem[result, e, esize] = SignedSat(element3 - product, esize);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

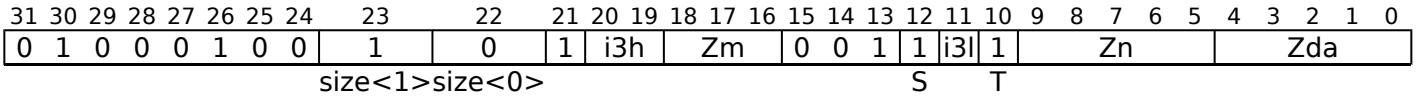
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQDMLSLT (indexed)

Signed saturating doubling multiply-subtract long from accumulator (top, indexed)

Multiply then double the odd-numbered signed elements within each 128-bit segment of the first source vector and the specified signed element in the corresponding second source vector segment. Each intermediate value is saturated to the double-width N-bit value's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Then destructively subtract from the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$.
The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.
It has encodings from 2 classes: 32-bit and 64-bit

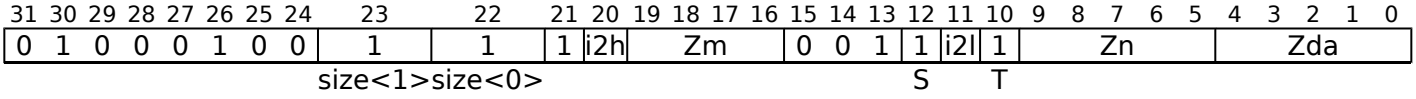
32-bit



SQDMLSLT <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
```

64-bit



SQDMLSLT <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field. For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields. For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel,   esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    integer element3 = SInt(Elem[result, e, 2*esize]);
    integer product = SInt(SignedSat(2 * element1 * element2, 2*esize));
    integer res = element3 - product;
    Elem[result, e, 2*esize] = SignedSat(res, 2*esize);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

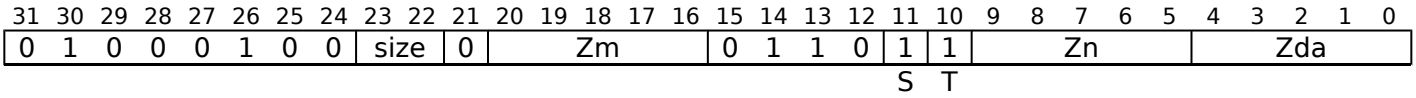
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQDMLSLT (vectors)

Signed saturating doubling multiply-subtract long from accumulator (top)

Multiply then double the corresponding odd-numbered signed elements of the first and second source vectors. Each intermediate value is saturated to the double-width N-bit value's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Then destructively subtract from the overlapping double-width elements of the addend and destination vector. Each destination element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.



SQDMLSLT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel1 = 1;
integer sel2 = 1;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2 * e + sel1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2 * e + sel2, esize DIV 2]);
    integer element3 = SInt(Elem[result, e, esize]);
    integer product = SInt(SignedSat(2 * element1 * element2, esize));
    Elem[result, e, esize] = SignedSat(element3 - product, esize);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

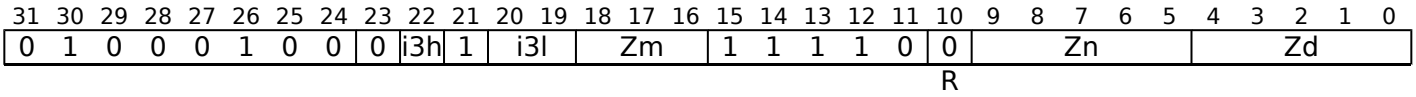
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQDMULH (indexed)

Signed saturating doubling multiply high (indexed)

Multiply all signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment, double and place the most significant half of the result in the corresponding elements of the destination vector register. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$.
The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element.
It has encodings from 3 classes: 16-bit , 32-bit and 64-bit

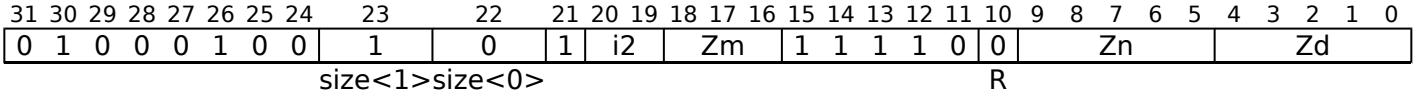
16-bit



SQDMULH <Zd>.H, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

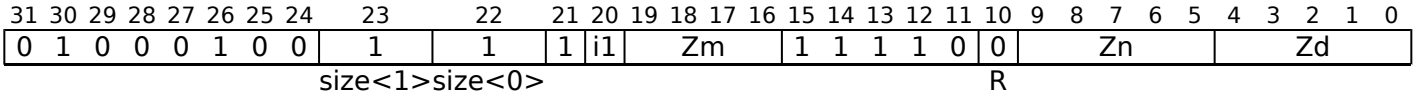
32-bit



SQDMULH <Zd>.S, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

64-bit



SQDMULH <Zd>.D, <Zn>.D, <Zm>.D[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the 16-bit and 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field. For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the 16-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields. For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field. For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, s, esize]);
    integer res = 2 * element1 * element2;
    Elem[result, e, esize] = SignedSat(res >> esize, esize);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

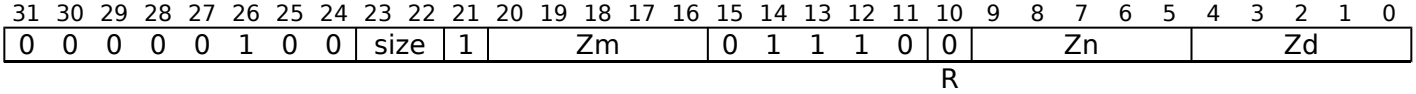
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQDMULH (vectors)

Signed saturating doubling multiply high (unpredicated)

Multiply then double the corresponding signed elements of the first and second source vectors, and place the most significant half of the results in the corresponding elements of the destination vector register. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.



SQDMULH <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    integer res = 2 * element1 * element2;
    Elem[result, e, esize] = SignedSat(res >> esize, esize);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

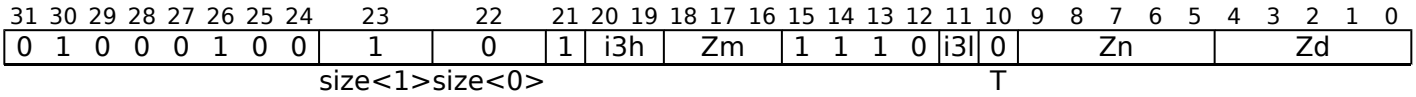
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQDMULLB (indexed)

Signed saturating doubling multiply long (bottom, indexed)

Multiply then double the even-numbered signed elements within each 128-bit segment of the first source vector and the specified element in the corresponding second source vector segment, and place the results in overlapping double-width elements of the destination vector register. Each result element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$.
The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.
It has encodings from 2 classes: 32-bit and 64-bit

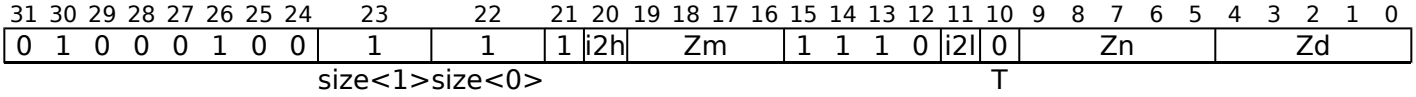
32-bit



SQDMULLB <Zd>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 0;
```

64-bit



SQDMULLB <Zd>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 0;
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field. For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields. For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation

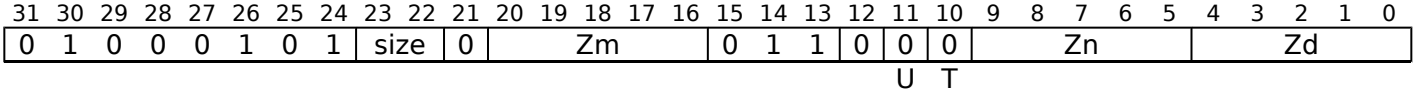
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel,   esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    integer res = 2 * element1 * element2;
    Elem[result, e, 2*esize] = SignedSat(res, 2*esize);

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQDMULLB (vectors)

Signed saturating doubling multiply long (bottom)

Multiply the corresponding even-numbered signed elements of the first and second source vectors, double and place the results in the overlapping double-width elements of the destination vector. Each result element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.



SQDMULLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

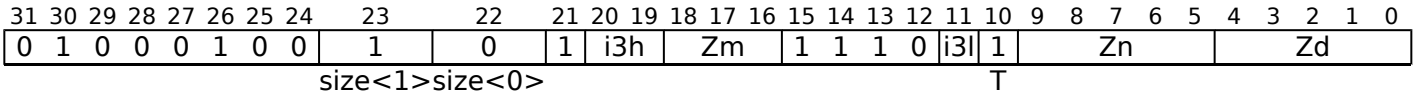
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2*e + 0, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    integer res = 2 * element1 * element2;
    Elem[result, e, esize] = SignedSat(res, esize);

Z[d, VL] = result;
```

SQDMULLT (indexed)

Signed saturating doubling multiply long (top, indexed)

Multiply then double the odd-numbered signed elements within each 128-bit segment of the first source vector and the specified element in the corresponding second source vector segment, and place the results in overlapping double-width elements of the destination vector register. Each result element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$.
The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.
It has encodings from 2 classes: 32-bit and 64-bit

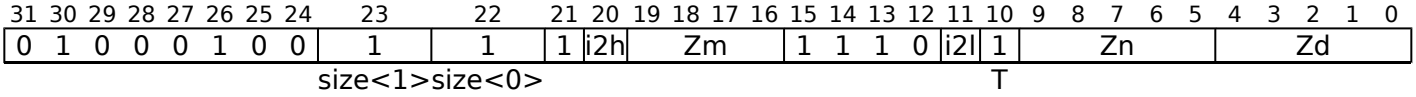
32-bit



SQDMULLT <Zd>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 1;
```

64-bit



SQDMULLT <Zd>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 1;
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field. For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields. For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation

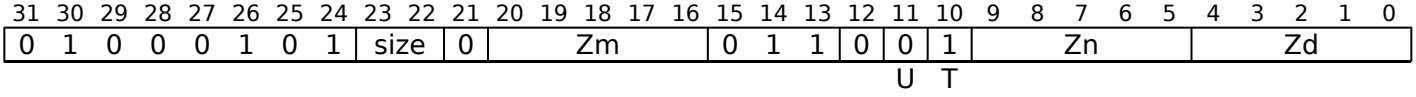
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = SInt(Elem[operand1, 2 * e + sel,   esize]);
    integer element2 = SInt(Elem[operand2, 2 * s + index, esize]);
    integer res = 2 * element1 * element2;
    Elem[result, e, 2*esize] = SignedSat(res, 2*esize);

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQDMULLT (vectors)

Signed saturating doubling multiply long (top)

Multiply the corresponding odd-numbered signed elements of the first and second source vectors, double and place the results in the overlapping double-width elements of the destination vector. Each result element is saturated to the double-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.



SQDMULLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, 2*e + 1, esize DIV 2]);
    integer element2 = SInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    integer res = 2 * element1 * element2;
    Elem[result, e, esize] = SignedSat(res, esize);

Z[d, VL] = result;
```

SQINCB

Signed saturating increment scalar by multiple of 8-bit predicate constraint element count

Determines the number of active 8-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

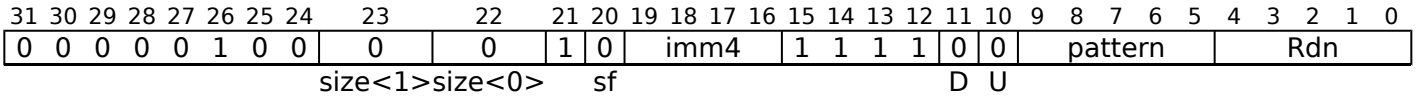
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

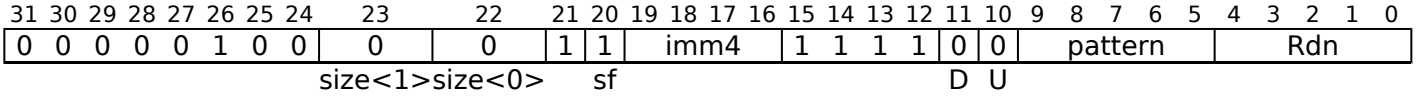
32-bit



SQINCB <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;
```

64-bit



SQINCB <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;
```

Assembler Symbols

<Xdn>	Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Wdn>	Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<pattern>	Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQINCD (scalar)

Signed saturating increment scalar by multiple of 64-bit predicate constraint element count

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

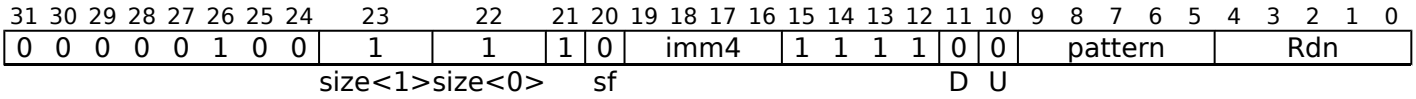
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

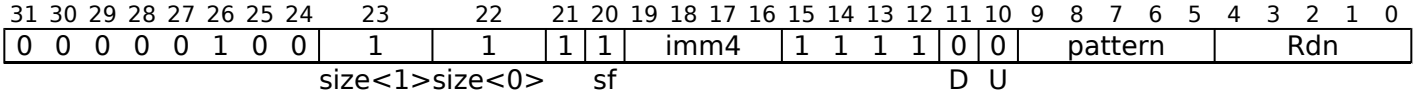
32-bit



SQINCD <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;
```

64-bit



SQINCD <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;
```

Assembler Symbols

<Xdn>	Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Wdn>	Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<pattern>	Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQINCD (vector)

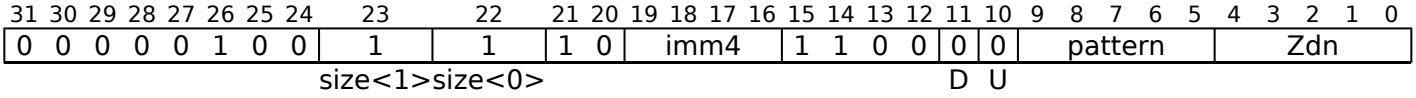
Signed saturating increment vector by multiple of 64-bit predicate constraint element count

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 64-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.



SQINCD <Zdn>.D{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + (count * imm), esize, unsigned);

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQINCH (scalar)

Signed saturating increment scalar by multiple of 16-bit predicate constraint element count

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

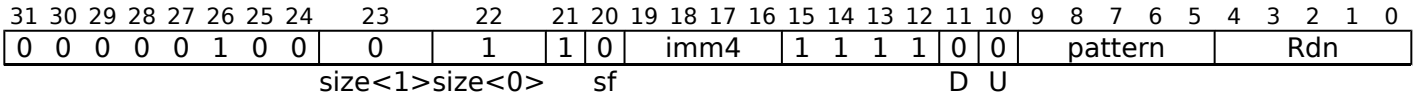
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

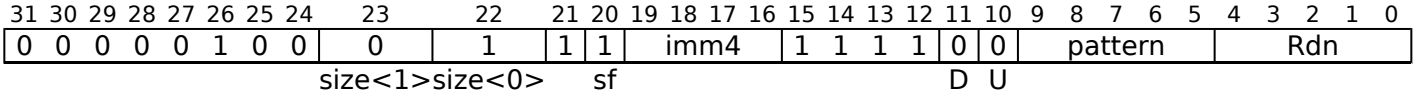
32-bit



SQINCH <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;
```

64-bit



SQINCH <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;
```

Assembler Symbols

<Xdn>	Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Wdn>	Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<pattern>	Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQINCH (vector)

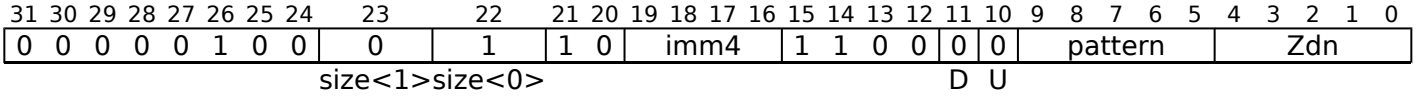
Signed saturating increment vector by multiple of 16-bit predicate constraint element count

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 16-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.



SQINCH <Zdn>.H{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + (count * imm), esize, unsigned);

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

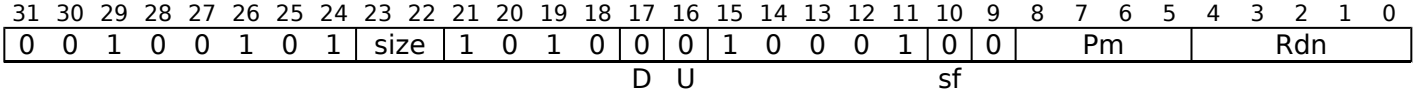
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQINCP (scalar)

Signed saturating increment scalar by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to increment the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.
It has encodings from 2 classes: 32-bit and 64-bit

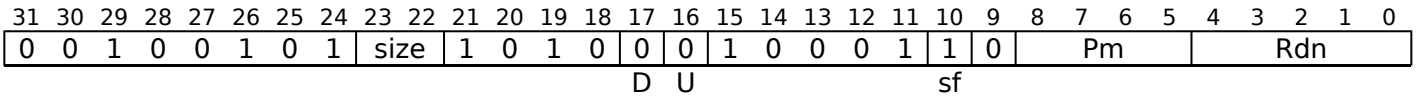
32-bit



SQINCP <Xdn>, <Pm>.<T>, <Wdn>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = FALSE;
integer ssize = 32;
```

64-bit



SQINCP <Xdn>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = FALSE;
integer ssize = 64;
```

Assembler Symbols

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(ssize) operand1 = X[dn, ssize];
bits(PL) operand2 = P[m, PL];
bits(ssize) result;
integer count = 0;

for e = 0 to elements-1
    if ElemP[operand2, e, esize] == '1' then
        count = count + 1;

integer element = Int(operand1, unsigned);
(result, -) = SatQ(element + count, ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
```

Operational information

If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

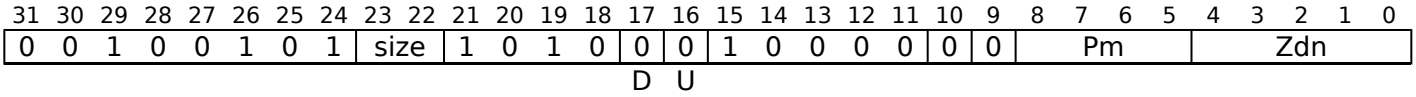
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQINCP (vector)

Signed saturating increment vector by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to increment all destination vector elements. The results are saturated to the element signed integer range.

The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.



SQINCP <Zdn>.<T>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) operand2 = P[m, PL];
bits(VL) result;
integer count = 0;

for e = 0 to elements-1
    if ElemP[operand2, e, esize] == '1' then
        count = count + 1;

for e = 0 to elements-1
    integer element = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element + count, esize, unsigned);

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQINCW (scalar)

Signed saturating increment scalar by multiple of 32-bit predicate constraint element count

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the source general-purpose register's signed integer range. A 32-bit saturated result is then sign-extended to 64 bits.

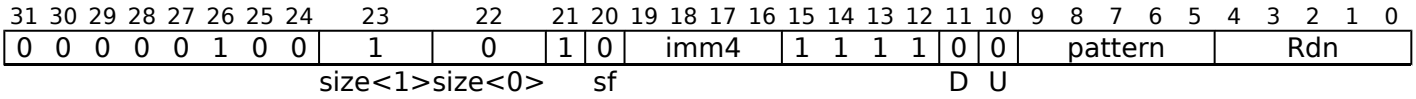
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

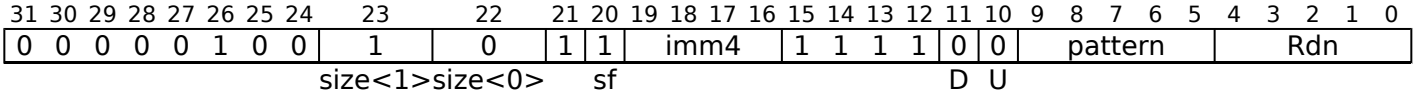
32-bit



SQINCW <Xdn>, <Wdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 32;
```

64-bit



SQINCW <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
integer ssize = 64;
```

Assembler Symbols

<Xdn>	Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Wdn>	Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<pattern>	Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQINCW (vector)

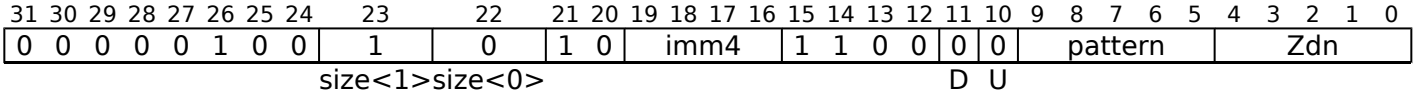
Signed saturating increment vector by multiple of 32-bit predicate constraint element count

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 32-bit signed integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.



SQINCW <Zdn>.S{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + (count * imm), esize, unsigned);

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

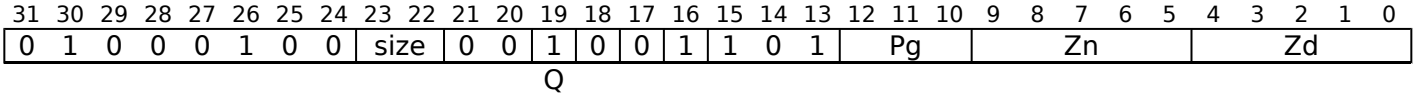
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQNEG

Signed saturating negate

Negate the signed integer value in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Inactive elements in the destination vector register remain unmodified.



SQNEG <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element = SInt(Elem[operand, e, esize]);
        element = -element;
        Elem[result, e, esize] = SignedSat(element, esize);

Z[d, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

SQRDCMLAH (indexed)

Saturating rounding doubling complex integer multiply-add high with rotate (indexed)

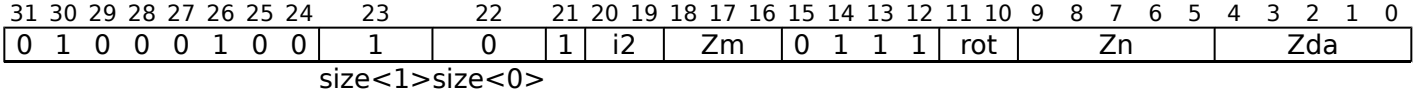
Multiply without saturation the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the integral numbers in each 128-bit segment of the first source vector by the specified complex number in the corresponding the second source vector segment rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation. Then double and add the products to the corresponding components of the complex numbers in the addend vector. Destructively place the most significant rounded half of the results in the corresponding elements of the addend vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.

These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.

Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.

It has encodings from 2 classes: 16-bit and 32-bit

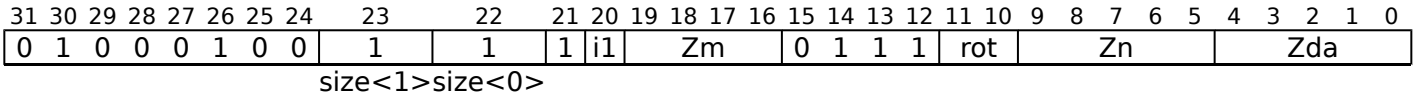
16-bit



SQRDCMLAH <Zda>.H, <Zn>.H, <Zm>.H[<imm>], <const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_r = (rot<0> != rot<1>);
boolean sub_i = (rot<1> == '1');
```

32-bit



SQRDCMLAH <Zda>.S, <Zn>.S, <Zm>.S[<imm>], <const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_r = (rot<0> != rot<1>);
boolean sub_i = (rot<1> == '1');
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> For the 16-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.

For the 32-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.

<imm> For the 16-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.

For the 32-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

<const> Is the const specifier, encoded in “rot”:

rot	<const>
00	#0
01	#90
10	#180
11	#270

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (2 * esize);
constant integer pairspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

integer round_const = 1 << (esize-1);
integer res_r, res_i;

for p = 0 to pairs-1
    integer segmentbase = p - (p MOD pairspersegment);
    integer s = segmentbase + index;
    integer elt1_a = SInt(Elem[operand1, 2 * p + sel_a, esize]);
    integer elt2_a = SInt(Elem[operand2, 2 * s + sel_a, esize]);
    integer elt2_b = SInt(Elem[operand2, 2 * s + sel_b, esize]);
    bits(esize) elt3_r = Elem[operand3, 2 * p + 0, esize];
    bits(esize) elt3_i = Elem[operand3, 2 * p + 1, esize];
    integer product_r =  elt1_a * elt2_a;
    integer product_i =  elt1_a * elt2_b;
    if sub_r then
        res_r = (SInt(elt3_r) << esize) - 2 * product_r + round_const;
    else
        res_r = (SInt(elt3_r) << esize) + 2 * product_r + round_const;
    if sub_i then
        res_i = (SInt(elt3_i) << esize) - 2 * product_i + round_const;
    else
        res_i = (SInt(elt3_i) << esize) + 2 * product_i + round_const;
    Elem[result, 2 * p + 0, esize] = SignedSat(res_r >> esize, esize);
    Elem[result, 2 * p + 1, esize] = SignedSat(res_i >> esize, esize);

Z[da, VL] = result;
```

Operational information

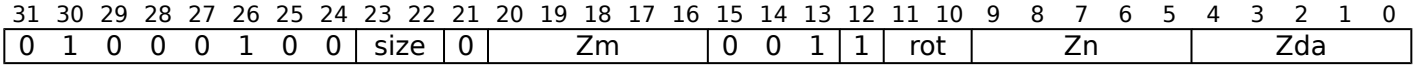
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

SQRDCMLAH (vectors)

Saturating rounding doubling complex integer multiply-add high with rotate

Multiply without saturation the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the integral numbers in the first source vector by the corresponding complex number in the second source vector rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation.
Then double and add the products to the corresponding components of the complex numbers in the addend vector. Destructively place the most significant rounded half of the results in the corresponding elements of the addend vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.
These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.
Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.



SQRDCMLAH <Zda>.<T>, <Zn>.<T>, <Zm>.<T>, <const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_r = (rot<0> != rot<1>);
boolean sub_i = (rot<1> == '1');
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<const> Is the const specifier, encoded in “rot”:

rot	<const>
00	#0
01	#90
10	#180
11	#270

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

integer round_const = 1 << (esize-1);
integer res_r, res_i;

for p = 0 to pairs-1
    integer elt1_a = SInt(Elem[operand1, 2 * p + sel_a, esize]);
    integer elt2_a = SInt(Elem[operand2, 2 * p + sel_a, esize]);
    integer elt2_b = SInt(Elem[operand2, 2 * p + sel_b, esize]);
    bits(esize) elt3_r = Elem[operand3, 2 * p + 0, esize];
    bits(esize) elt3_i = Elem[operand3, 2 * p + 1, esize];
    integer product_r =  elt1_a * elt2_a;
    integer product_i =  elt1_a * elt2_b;
    if sub_r then
        res_r = (SInt(elt3_r) << esize) - 2 * product_r + round_const;
    else
        res_r = (SInt(elt3_r) << esize) + 2 * product_r + round_const;
    if sub_i then
        res_i = (SInt(elt3_i) << esize) - 2 * product_i + round_const;
    else
        res_i = (SInt(elt3_i) << esize) + 2 * product_i + round_const;
    Elem[result, 2 * p + 0, esize] = SignedSat(res_r >> esize, esize);
    Elem[result, 2 * p + 1, esize] = SignedSat(res_i >> esize, esize);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

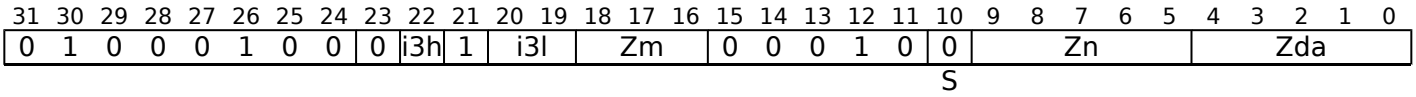
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQRDMLAH (indexed)

Signed saturating rounding doubling multiply-add high to accumulator (indexed)

Multiply then double all signed elements within each 128-bit segment of the first source vector and the specified signed element of the corresponding second source vector segment, and destructively add the rounded high half of each result to the corresponding elements of the addend and destination vector. Each destination element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$.
The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element.
It has encodings from 3 classes: 16-bit , 32-bit and 64-bit

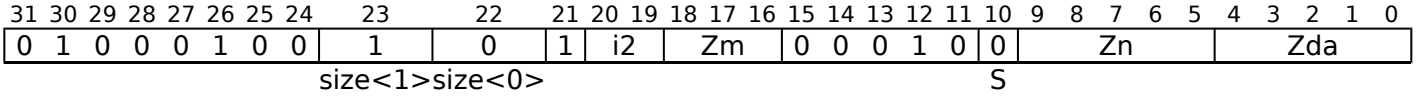
16-bit



SQRDMLAH <Zda>.H, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

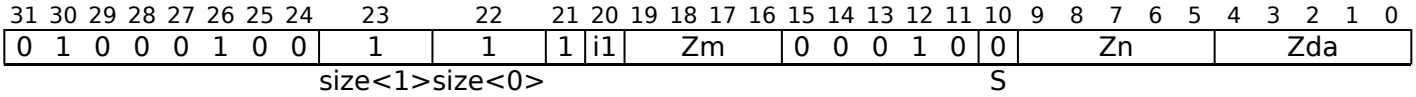
32-bit



SQRDMLAH <Zda>.S, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

64-bit



SQRDMLAH <Zda>.D, <Zn>.D, <Zm>.D[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the 16-bit and 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field. For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the 16-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields. For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field. For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
constant integer round_const = 1 << (esize - 1);

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, s, esize]);
    integer element3 = SInt(Elem[operand3, e, esize]);
    integer res = (element3 << esize) + (2 * element1 * element2);
    Elem[result, e, esize] = SignedSat((res + round_const) >> esize, esize);

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

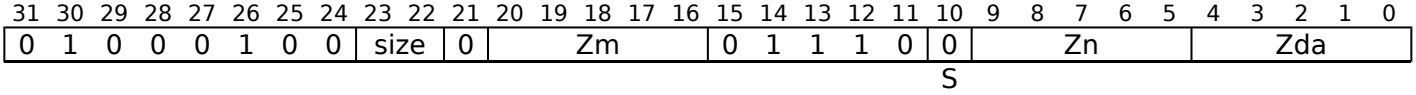
- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQRDMLAH (vectors)

Signed saturating rounding doubling multiply-add high to accumulator (unpredicated)

Multiply then double the corresponding signed elements of the first and second source vectors, and destructively add the rounded high half of each result to the corresponding elements of the addend and destination vector. Each destination element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.



SQRDMLAH <Zda>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
constant integer round_const = 1 << (esize - 1);

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    integer element3 = SInt(Elem[operand3, e, esize]);
    integer res = (element3 << esize) + (2 * element1 * element2);
    Elem[result, e, esize] = SignedSat((res + round_const) >> esize, esize);

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

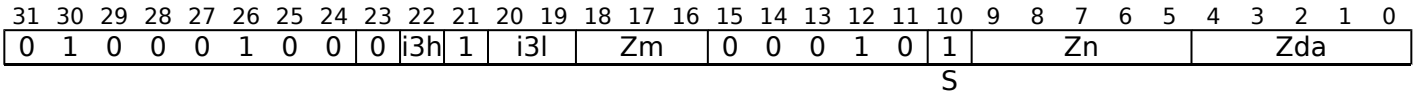
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQRDMLSH (indexed)

Signed saturating rounding doubling multiply-subtract high from accumulator (indexed)

Multiply then double all signed elements within each 128-bit segment of the first source vector and the specified signed element of the corresponding second source vector segment, and destructively subtract the rounded high half of each result to the corresponding elements of the addend and destination vector. Each destination element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$.
The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element.
It has encodings from 3 classes: 16-bit , 32-bit and 64-bit

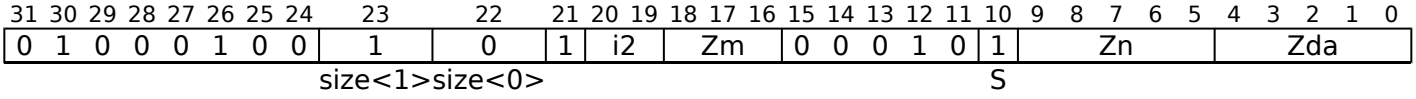
16-bit



SQRDMLSH <Zda>.H, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

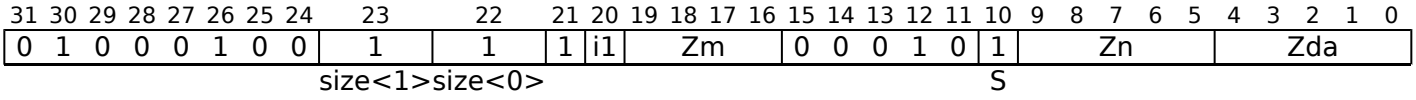
32-bit



SQRDMLSH <Zda>.S, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

64-bit



SQRDMLSH <Zda>.D, <Zn>.D, <Zm>.D[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the 16-bit and 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field. For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the 16-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields. For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field. For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
constant integer round_const = 1 << (esize - 1);

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, s, esize]);
    integer element3 = SInt(Elem[operand3, e, esize]);
    integer res = (element3 << esize) - (2 * element1 * element2);
    Elem[result, e, esize] = SignedSat((res + round_const) >> esize, esize);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

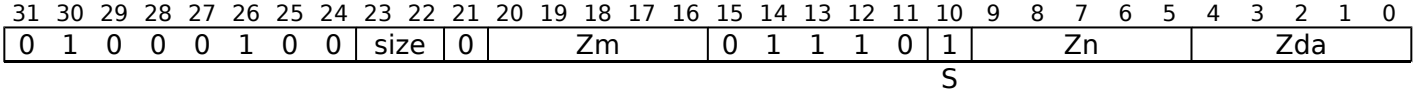
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQRDMLSH (vectors)

Signed saturating rounding doubling multiply-subtract high from accumulator (unpredicated)

Multiply then double the corresponding signed elements of the first and second source vectors, and destructively subtract the rounded high half of each result from the corresponding elements of the addend and destination vector. Each destination element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.



SQRDMLSH <Zda>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;
constant integer round_const = 1 << (esize - 1);

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    integer element3 = SInt(Elem[operand3, e, esize]);
    integer res = (element3 << esize) - (2 * element1 * element2);
    Elem[result, e, esize] = SignedSat((res + round_const) >> esize, esize);

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

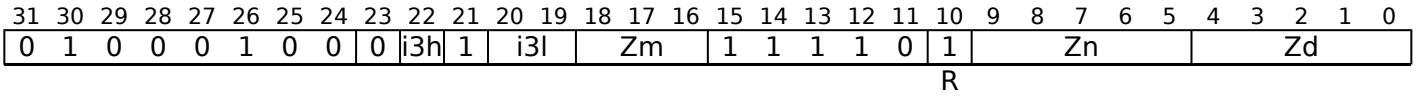
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQRDMULH (indexed)

Signed saturating rounding doubling multiply high (indexed)

Multiply all signed elements within each 128-bit segment of the first source vector by the specified signed element in the corresponding second source vector segment, double and place the most significant rounded half of the result in the corresponding elements of the destination vector register. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$.
The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 1 to 3 bits depending on the size of the element.
It has encodings from 3 classes: 16-bit , 32-bit and 64-bit

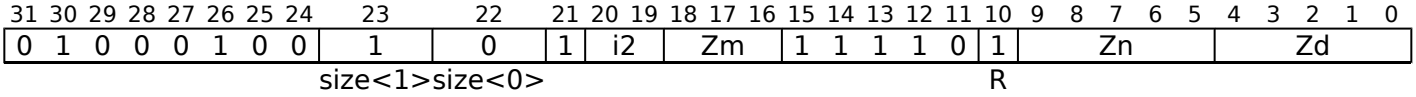
16-bit



SQRDMULH <Zd>.H, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

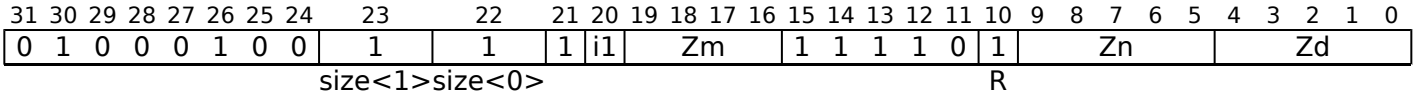
32-bit



SQRDMULH <Zd>.S, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

64-bit



SQRDMULH <Zd>.D, <Zn>.D, <Zm>.D[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the 16-bit and 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field. For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the 16-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields. For the 32-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field. For the 64-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
constant integer round_const = 1 << (esize - 1);

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, s, esize]);
    integer res = 2 * element1 * element2;
    Elem[result, e, esize] = SignedSat((res + round_const) >> esize, esize);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

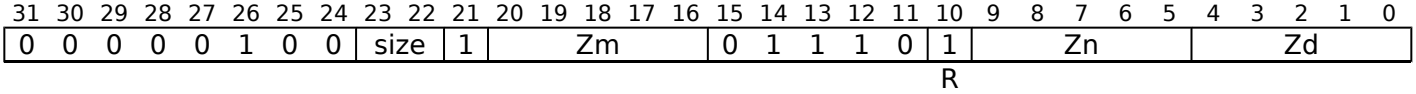
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQRDMULH (vectors)

Signed saturating rounding doubling multiply high (unpredicated)

Multiply then double the corresponding signed elements of the first and second source vectors, and place the most significant rounded half of the result in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.



SQRDMULH <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
constant integer round_const = 1 << (esize - 1);

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    integer res = 2 * element1 * element2;
    Elem[result, e, esize] = SignedSat((res + round_const) >> esize, esize);

Z[d, VL] = result;
```

Operational information

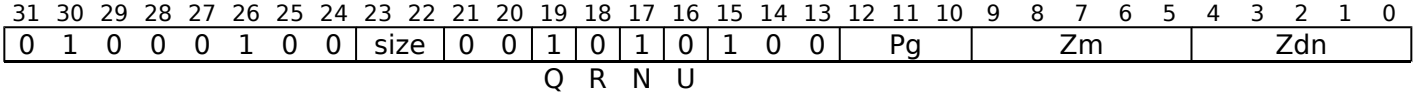
If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

SQRSHL

Signed saturating rounding shift left by vector (predicated)

Shift active signed elements of the first source vector by corresponding elements of the second source vector and destructively place the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Inactive elements in the destination vector register remain unmodified.



SQRSHL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element = SInt(Elem[operand1, e, esize]);
        integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
        integer round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
        integer res = (element + round_const) << shift;
        Elem[result, e, esize] = SignedSat(res, esize);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

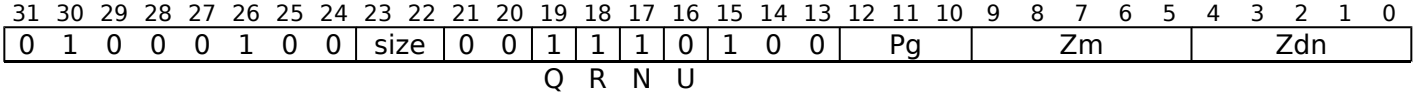
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQRSHLR

Signed saturating rounding shift left reversed vectors (predicated)

Shift active signed elements of the second source vector by corresponding elements of the first source vector and destructively place the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Inactive elements in the destination vector register remain unmodified.



SQRSHLR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand2 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element = SInt(Elem[operand1, e, esize]);
        integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
        integer round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
        integer res = (element + round_const) << shift;
        Elem[result, e, esize] = SignedSat(res, esize);
    else
        Elem[result, e, esize] = Elem[operand2, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

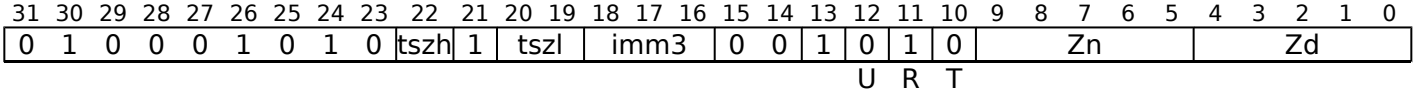
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQRSHRNB

Signed saturating rounding shift right narrow by immediate (bottom)

Shift each signed integer value in the source vector elements right by an immediate value, and place the rounded results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. Each result element is saturated to the half-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



SQRSHRNB <Zd>.<T>, <Zn>.<Tb>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '000' UNDEFINED;
    when '001' esize = 8;
    when '01x' esize = 16;
    when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

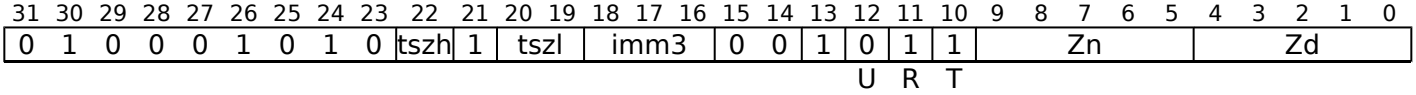
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;
integer round_const = 1 << (shift-1);

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = (SInt(element) + round_const) >> shift;
    Elem[result, 2*e + 0, esize] = SignedSat(res, esize);
    Elem[result, 2*e + 1, esize] = Zeros(esize);

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQRSHRNT

Signed saturating rounding shift right narrow by immediate (top)

Shift each signed integer value in the source vector elements right by an immediate value, and place the rounded results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. Each result element is saturated to the half-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



SQRSHRNT <Zd>.<T>, <Zn>.<Tb>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '000' UNDEFINED;
    when '001' esize = 8;
    when '01x' esize = 16;
    when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

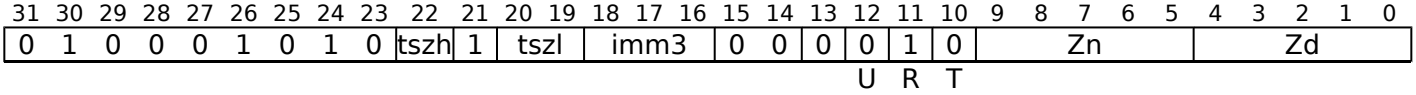
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result = Z[d, VL];
integer round_const = 1 << (shift-1);

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = (SInt(element) + round_const) >> shift;
    Elem[result, 2*e + 1, esize] = SignedSat(res, esize);

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQRSHRUNB

Signed saturating rounding shift right unsigned narrow by immediate (bottom)

Shift each signed integer value in the source vector elements right by an immediate value, and place the rounded results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



SQRSHRUNB <Zd>.<T>, <Zn>.<Tb>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '000' UNDEFINED;
    when '001' esize = 8;
    when '01x' esize = 16;
    when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

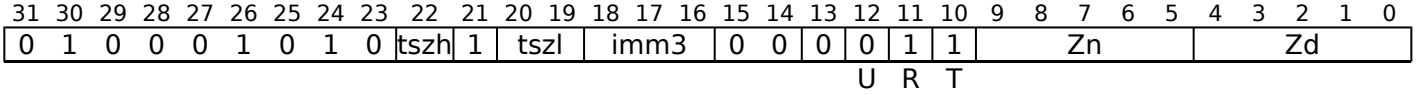
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;
integer round_const = 1 << (shift-1);

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = (SInt(element) + round_const) >> shift;
    Elem[result, 2*e + 0, esize] = UnsignedSat(res, esize);
    Elem[result, 2*e + 1, esize] = Zeros(esize);

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQRSHRUNT

Signed saturating rounding shift right unsigned narrow by immediate (top)

Shift each signed integer value in the source vector elements right by an immediate value, and place the rounded results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



SQRSHRUNT <Zd>.<T>, <Zn>.<Tb>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '000' UNDEFINED;
    when '001' esize = 8;
    when '01x' esize = 16;
    when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

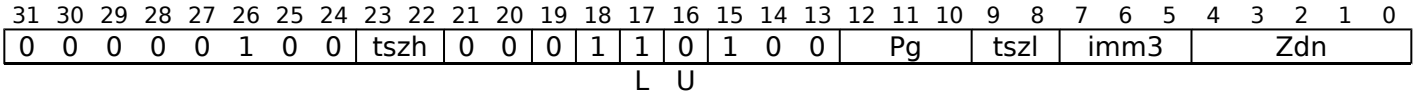
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result = Z[d, VL];
integer round_const = 1 << (shift-1);

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = (SInt(element) + round_const) >> shift;
    Elem[result, 2*e + 1, esize] = UnsignedSat(res, esize);

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQSHL (immediate)

Signed saturating shift left by immediate

Shift left by immediate each active signed element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. Inactive elements in the destination vector register remain unmodified.



SQSHL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
case tsize of
    when '0000' UNDEFINED;
    when '0001' esize = 8;
    when '001x' esize = 16;
    when '01xx' esize = 32;
    when '1xxx' esize = 64;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = UInt(tsize:imm3) - esize;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) mask = P[g, PL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    if ElemP[mask, e, esize] == '1' then
        integer res = element1 << shift;
        Elem[result, e, esize] = SignedSat(res, esize);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

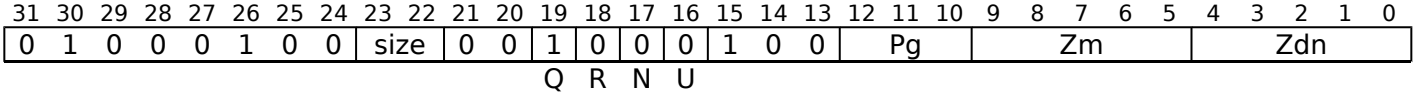
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQSHL (vectors)

Signed saturating shift left by vector (predicated)

Shift active signed elements of the first source vector by corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Inactive elements in the destination vector register remain unmodified.



SQSHL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element = SInt(Elem[operand1, e, esize]);
        integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
        integer res = element << shift;
        Elem[result, e, esize] = SignedSat(res, esize);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

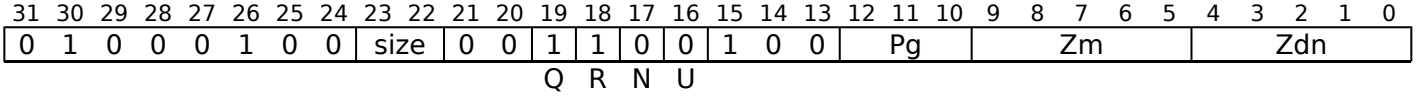
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQSHLR

Signed saturating shift left reversed vectors (predicated)

Shift active signed elements of the second source vector by corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Inactive elements in the destination vector register remain unmodified.



SQSHLR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand2 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element = SInt(Elem[operand1, e, esize]);
        integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
        integer res = element << shift;
        Elem[result, e, esize] = SignedSat(res, esize);
    else
        Elem[result, e, esize] = Elem[operand2, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

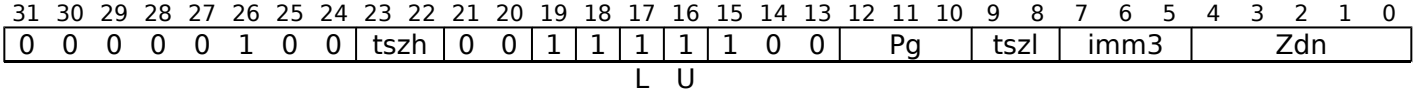
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQSHLU

Signed saturating shift left unsigned by immediate

Shift left by immediate each active signed element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. Inactive elements in the destination vector register remain unmodified.



SQSHLU <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
case tsize of
    when '0000' UNDEFINED;
    when '0001' esize = 8;
    when '001x' esize = 16;
    when '01xx' esize = 32;
    when '1xxx' esize = 64;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = UInt(tsize:imm3) - esize;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) mask = P[g, PL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    if ElemP[mask, e, esize] == '1' then
        integer res = element1 << shift;
        Elem[result, e, esize] = UnsignedSat(res, esize);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

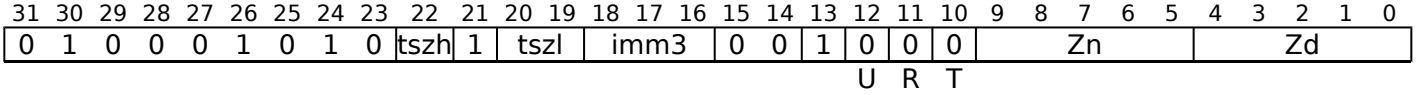
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQSHRNB

Signed saturating shift right narrow by immediate (bottom)

Shift each signed integer value in the source vector elements right by an immediate value, and place the truncated results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. Each result element is saturated to the half-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



SQSHRNB <Zd>.<T>, <Zn>.<Tb>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '000' UNDEFINED;
    when '001' esize = 8;
    when '01x' esize = 16;
    when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

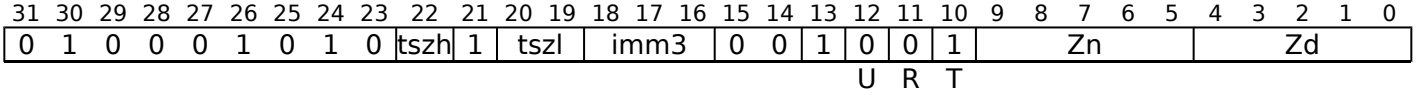
Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = SInt(element) >> shift;
    Elem[result, 2*e + 0, esize] = SignedSat(res, esize);
    Elem[result, 2*e + 1, esize] = Zeros(esize);

Z[d, VL] = result;
```

SQSHRNT

Signed saturating shift right narrow by immediate (top)

Shift each signed integer value in the source vector elements right by an immediate value, and place the truncated results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. Each result element is saturated to the half-width N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



SQSHRNT <Zd>.<T>, <Zn>.<Tb>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '000' UNDEFINED;
    when '001' esize = 8;
    when '01x' esize = 16;
    when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

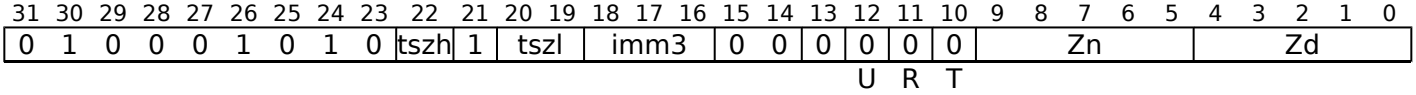
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = SInt(element) >> shift;
    Elem[result, 2*e + 1, esize] = SignedSat(res, esize);

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQSHRUNB

Signed saturating shift right unsigned narrow by immediate (bottom)

Shift each signed integer value in the source vector elements right by an immediate value, and place the truncated results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



SQSHRUNB <Zd>.<T>, <Zn>.<Tb>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '000' UNDEFINED;
    when '001' esize = 8;
    when '01x' esize = 16;
    when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

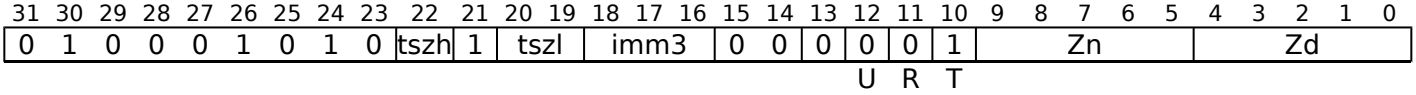
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = SInt(element) >> shift;
    Elem[result, 2*e + 0, esize] = UnsignedSat(res, esize);
    Elem[result, 2*e + 1, esize] = Zeros(esize);

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQSHRUNT

Signed saturating shift right unsigned narrow by immediate (top)

Shift each signed integer value in the source vector elements right by an immediate value, and place the truncated results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



SQSHRUNT <Zd>.<T>, <Zn>.<Tb>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '000' UNDEFINED;
    when '001' esize = 8;
    when '01x' esize = 16;
    when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

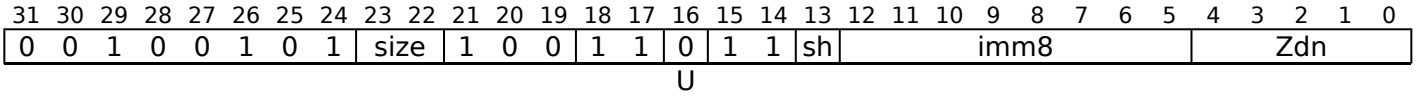
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = SInt(element) >> shift;
    Elem[result, 2*e + 1, esize] = UnsignedSat(res, esize);

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQSUB (immediate)

Signed saturating subtract immediate (unpredicated)

Signed saturating subtract of an unsigned immediate from each element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.
The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.
The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<uimm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".



SQSUB <Zdn>.<T>, <Zdn>.<T>, #<imm>{, <shift>}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = UInt(imm8);
if sh == '1' then imm = imm << 8;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<imm> Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #8

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - imm, esize, unsigned);

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

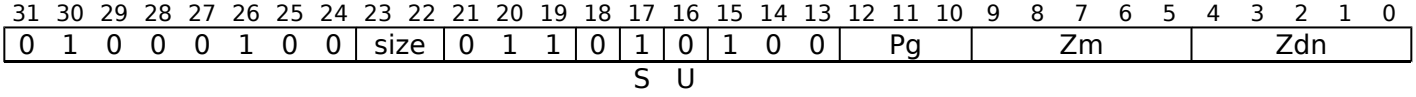
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQSUB (vectors, predicated)

Signed saturating subtraction (predicated)

Subtract active signed elements of the second source vector from corresponding signed elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Inactive elements in the destination vector register remain unmodified.



SQSUB <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    if ElemP[mask, e, esize] == '1' then
        integer res = SInt(Sat(element1 - element2, esize, unsigned));
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

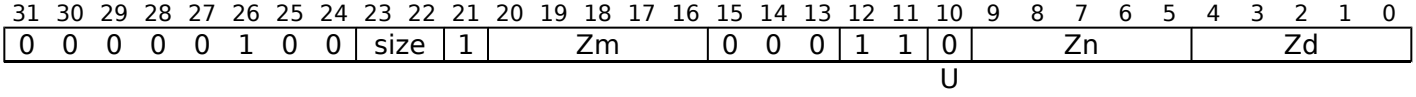
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQSUB (vectors, unpredicated)

Signed saturating subtract vectors (unpredicated)

Signed saturating subtract all elements of the second source vector from corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.



SQSUB <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean unsigned = FALSE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

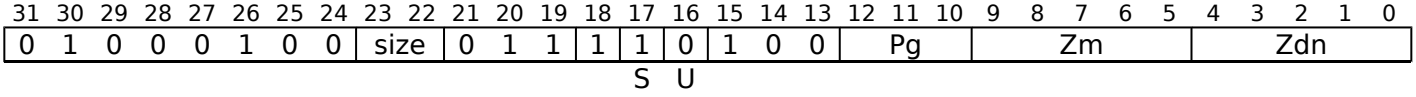
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - element2, esize, unsigned);

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQSUBR

Signed saturating subtraction reversed vectors (predicated)

Subtract active signed elements of the first source vector from corresponding signed elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Inactive elements in the destination vector register remain unmodified.



SQSUBR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = FALSE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    if ElemP[mask, e, esize] == '1' then
        integer res = SInt(Sat(element2 - element1, esize, unsigned));
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

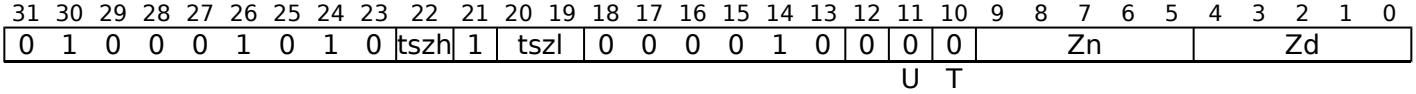
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQXTNB

Signed saturating extract narrow (bottom)

Saturate the signed integer value in each source element to half the original source element width, and place the results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero.



SQXTNB <Zd>.<T>, <Zn>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '001' esize = 16;
    when '010' esize = 32;
    when '100' esize = 64;
    otherwise UNDEFINED;
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	10	H
x	11	RESERVED
1	00	S
1	01	RESERVED
1	10	RESERVED

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	10	S
x	11	RESERVED
1	00	D
1	01	RESERVED
1	10	RESERVED

Operation

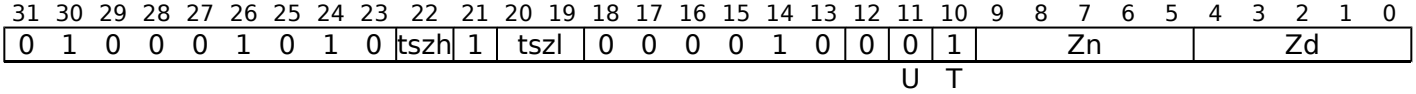
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) result;
constant integer halfesize = esize DIV 2;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    bits(halfesize) res = SignedSat(element1, halfesize);
    Elem[result, 2*e + 0, halfesize] = res;
    Elem[result, 2*e + 1, halfesize] = Zeros(halfesize);

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQXTNT

Signed saturating extract narrow (top)

Saturate the signed integer value in each source element to half the original source element width, and place the results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged.



SQXTNT <Zd>.<T>, <Zn>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '001' esize = 16;
    when '010' esize = 32;
    when '100' esize = 64;
    otherwise UNDEFINED;
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	10	H
x	11	RESERVED
1	00	S
1	01	RESERVED
1	10	RESERVED

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	10	S
x	11	RESERVED
1	00	D
1	01	RESERVED
1	10	RESERVED

Operation

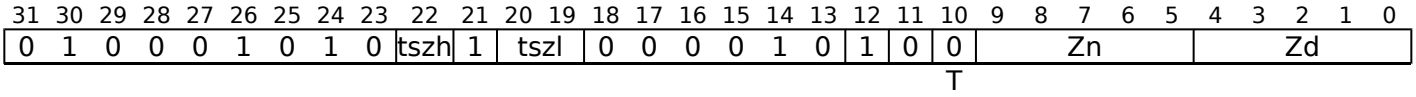
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) result = Z[d, VL];
constant integer halfesize = esize DIV 2;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    bits(halfesize) res = SignedSat(element1, halfesize);
    Elem[result, 2*e + 1, halfesize] = res;

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQXTUNB

Signed saturating unsigned extract narrow (bottom)

Saturate the signed integer value in each source element to an unsigned integer value that is half the original source element width, and place the results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero.



SQXTUNB <Zd>.<T>, <Zn>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '001' esize = 16;
    when '010' esize = 32;
    when '100' esize = 64;
    otherwise UNDEFINED;
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	10	H
x	11	RESERVED
1	00	S
1	01	RESERVED
1	10	RESERVED

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	10	S
x	11	RESERVED
1	00	D
1	01	RESERVED
1	10	RESERVED

Operation

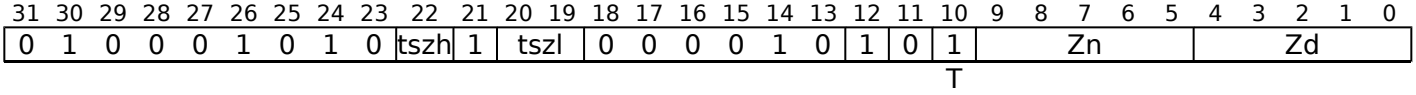
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) result;
constant integer halfesize = esize DIV 2;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    bits(halfesize) res = UnsignedSat(element1, halfesize);
    Elem[result, 2*e + 0, halfesize] = res;
    Elem[result, 2*e + 1, halfesize] = Zeros(halfesize);

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SQXTUNT

Signed saturating unsigned extract narrow (top)

Saturate the signed integer value in each source element to an unsigned integer value that is half the original source element width, and place the results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged.



SQXTUNT <Zd>.<T>, <Zn>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '001' esize = 16;
    when '010' esize = 32;
    when '100' esize = 64;
    otherwise UNDEFINED;
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	10	H
x	11	RESERVED
1	00	S
1	01	RESERVED
1	10	RESERVED

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	10	S
x	11	RESERVED
1	00	D
1	01	RESERVED
1	10	RESERVED

Operation

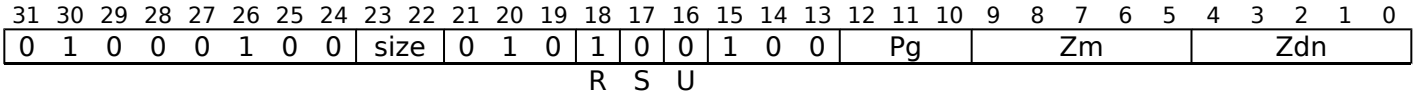
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) result = Z[d, VL];
constant integer halfesize = esize DIV 2;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    bits(halfesize) res = UnsignedSat(element1, halfesize);
    Elem[result, 2*e + 1, halfesize] = res;

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SRHADD

Signed rounding halving addition

Add active signed elements of the first source vector to corresponding signed elements of the second source vector, shift right one bit, and destructively place the rounded results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



SRHADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
integer round_const = 1;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, e, esize]);
    if ElemP[mask, e, esize] == '1' then
        integer res = (element1 + element2 + round_const) >> 1;
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

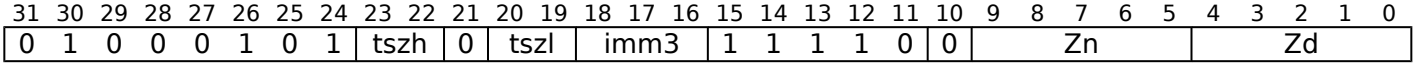
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SRI

Shift right and insert (immediate)

Shift each source vector element right by an immediate value, and insert the result into the corresponding vector element in the destination vector register, merging the shifted bits from each source element with existing bits in each destination vector element. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



SRI <Zd>.<T>, <Zn>.<T>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
case tsize of
    when '0000' UNDEFINED;
    when '0001' esize = 8;
    when '001x' esize = 16;
    when '01xx' esize = 32;
    when '1xxx' esize = 64;
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand = Z[n, VL];
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    bits(esize) element1 = Elem[result, e, esize];
    bits(esize) element2 = Elem[operand, e, esize];
    bits(esize) mask = LSR(Ones(esize), shift);
    bits(esize) shiftedval = LSR(element2, shift);
    Elem[result, e, esize] = (element1 AND (NOT mask)) OR shiftedval;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

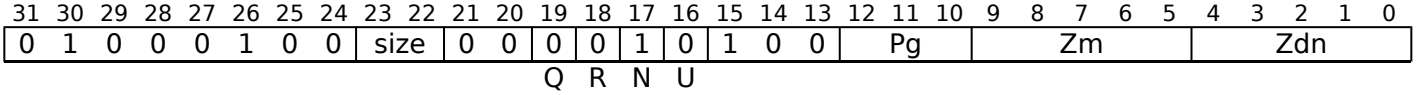
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SRSHL

Signed rounding shift left by vector (predicated)

Shift active signed elements of the first source vector by corresponding elements of the second source vector and destructively place the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Inactive elements in the destination vector register remain unmodified.



SRSHL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element = SInt(Elem[operand1, e, esize]);
        integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
        integer round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
        integer res = (element + round_const) << shift;
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

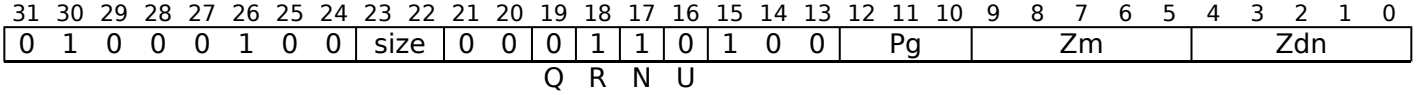
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SRSHLR

Signed rounding shift left reversed vectors (predicated)

Shift active signed elements of the second source vector by corresponding elements of the first source vector and destructively place the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Inactive elements in the destination vector register remain unmodified.



SRSHLR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand2 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element = SInt(Elem[operand1, e, esize]);
        integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
        integer round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
        integer res = (element + round_const) << shift;
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand2, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

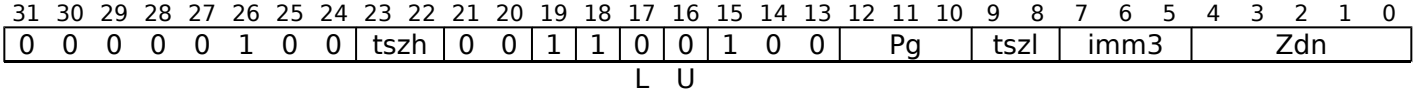
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SRSHR

Signed rounding shift right by immediate

Shift right by immediate each active signed element of the source vector, and destructively place the rounded results in the corresponding elements of the source vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. Inactive elements in the destination vector register remain unmodified.



SRSHR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
case tsize of
    when '0000' UNDEFINED;
    when '0001' esize = 8;
    when '001x' esize = 16;
    when '01xx' esize = 32;
    when '1xxx' esize = 64;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) mask = P[g, PL];
bits(VL) result;
integer round_const = 1 << (shift-1);

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    if ElemP[mask, e, esize] == '1' then
        integer res = (element1 + round_const) >> shift;
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

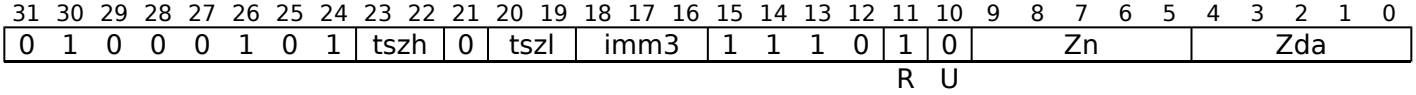
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SRSRA

Signed rounding shift right and accumulate (immediate)

Shift right by immediate each signed element of the source vector, preserving the sign bit, and add the rounded intermediate result destructively to the corresponding elements of the addend vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



SRSRA <Zda>.<T>, <Zn>.<T>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
case tsize of
    when '0000' UNDEFINED;
    when '0001' esize = 8;
    when '001x' esize = 16;
    when '01xx' esize = 32;
    when '1xxx' esize = 64;
integer n = UInt(Zn);
integer da = UInt(Zda);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[da, VL];
bits(VL) result;
integer round_const = 1 << (shift - 1);

for e = 0 to elements-1
    integer element = (SInt(Elem[operand1, e, esize]) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

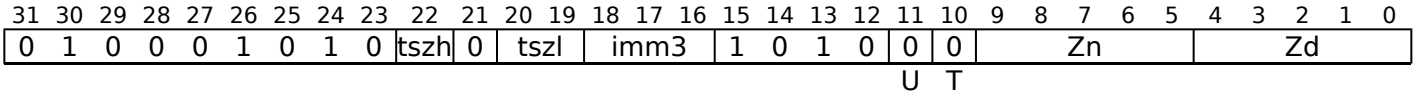
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SSHLLB

Signed shift left long by immediate (bottom)

Shift left by immediate each even-numbered signed element of the source vector, and place the results in the overlapping double-width elements of the destination vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. This instruction is unpredicated.



SSHLLB <Zd>.<T>, <Zn>.<Tb>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '000' UNDEFINED;
    when '001' esize = 8;
    when '01x' esize = 16;
    when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = UInt(tsize:imm3) - esize;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element = Elem[operand, 2*e + 0, esize];
    integer shifted_value = SInt(element) << shift;
    Elem[result, e, 2*esize] = shifted_value<2*esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

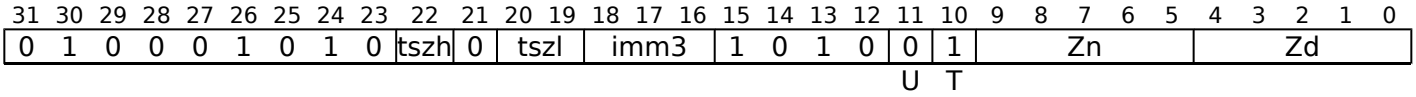
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SSHLLT

Signed shift left long by immediate (top)

Shift left by immediate each odd-numbered signed element of the source vector, and place the results in the overlapping double-width elements of the destination vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. This instruction is unpredicated.



SSHLLT <Zd>.<T>, <Zn>.<Tb>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '000' UNDEFINED;
    when '001' esize = 8;
    when '01x' esize = 16;
    when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = UInt(tsize:imm3) - esize;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element = Elem[operand, 2*e + 1, esize];
    integer shifted_value = SInt(element) << shift;
    Elem[result, e, 2*esize] = shifted_value<2*esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

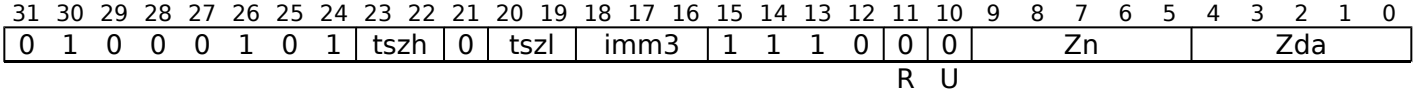
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SSRA

Signed shift right and accumulate (immediate)

Shift right by immediate each signed element of the source vector, preserving the sign bit, and add the truncated intermediate result destructively to the corresponding elements of the addend vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



SSRA <Zda>.<T>, <Zn>.<T>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
case tsize of
    when '0000' UNDEFINED;
    when '0001' esize = 8;
    when '001x' esize = 16;
    when '01xx' esize = 32;
    when '1xxx' esize = 64;
integer n = UInt(Zn);
integer da = UInt(Zda);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element = SInt(Elem[operand1, e, esize]) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.

    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

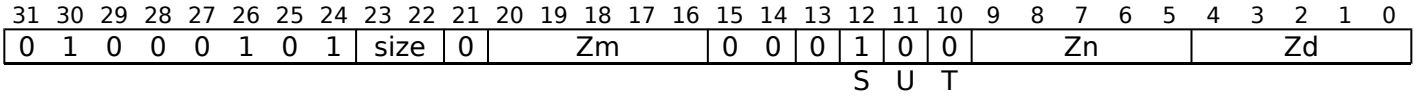
- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SSUBLB

Signed subtract long (bottom)

Subtract the even-numbered signed elements of the second source vector from the corresponding signed elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



SSUBLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 0;
integer sel2 = 0;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 - element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

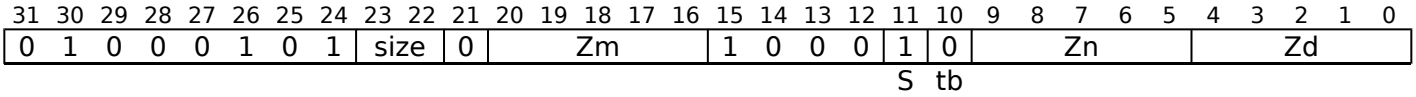
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SSUBLBT

Signed subtract long (bottom - top)

Subtract the odd-numbered signed elements of the second source vector from the even-numbered signed elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



SSUBLBT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 0;
integer sel2 = 1;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 - element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

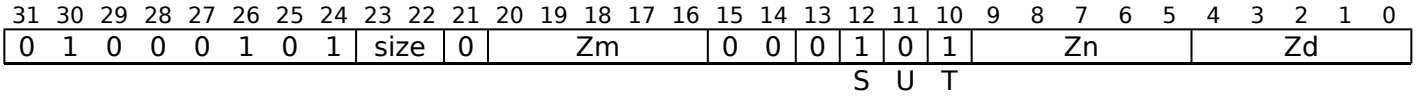
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SSUBLT

Signed subtract long (top)

Subtract the odd-numbered signed elements of the second source vector from the corresponding signed elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



SSUBLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 1;
integer sel2 = 1;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 - element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

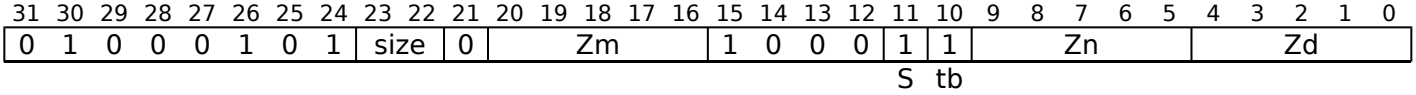
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SSUBLTB

Signed subtract long (top - bottom)

Subtract the even-numbered signed elements of the second source vector from the odd-numbered signed elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



SSUBLTB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 1;
integer sel2 = 0;
boolean unsigned = FALSE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 - element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

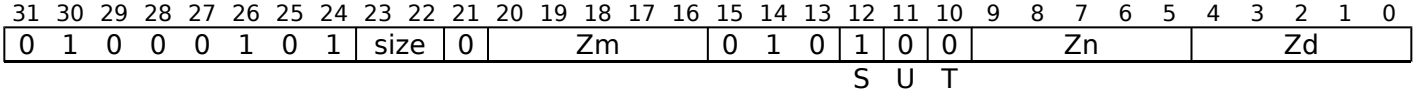
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SSUBWB

Signed subtract wide (bottom)

Subtract the even-numbered signed elements of the second source vector from the overlapping double-width elements of the first source vector and place the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.



SSUBWB <Zd>.<T>, <Zn>.<T>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    Elem[result, e, esize] = (element1 - element2)<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

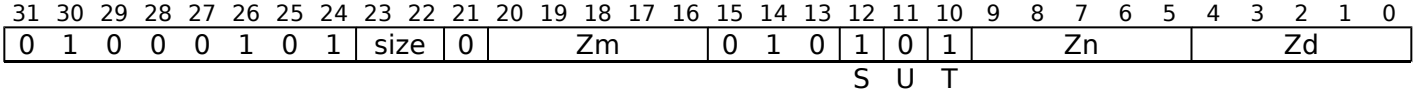
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SSUBWT

Signed subtract wide (top)

Subtract the even-numbered signed elements of the second source vector from the overlapping double-width elements of the first source vector and place the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.



SSUBWT <Zd>.<T>, <Zn>.<T>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = SInt(Elem[operand1, e, esize]);
    integer element2 = SInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    Elem[result, e, esize] = (element1 - element2)<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

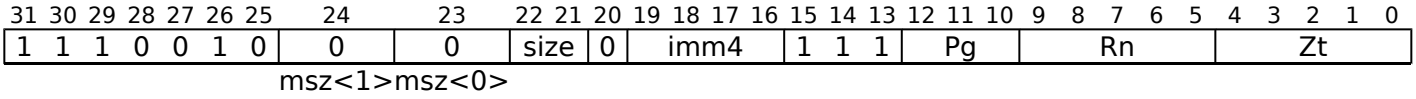
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST1B (scalar plus immediate)

Contiguous store bytes from vector (immediate index)

Contiguous store of bytes from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.



ST1B { <Zt>.<T> }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8 << UInt(size);
constant integer msize = 8;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

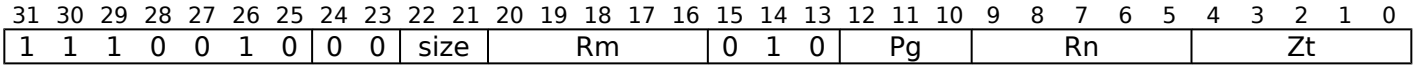
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) src;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST1B (scalar plus scalar)

Contiguous store bytes from vector (scalar index)

Contiguous store of bytes from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.



ST1B { <Zt>.<T> }, <Pg>, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8 << UInt(size);
constant integer msize = 8;
```

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
```

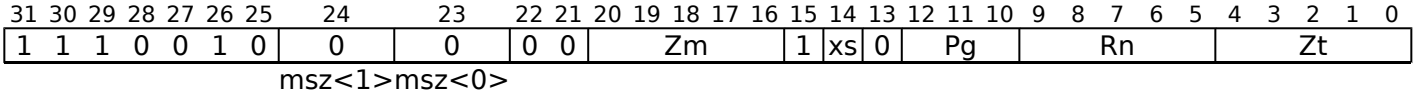
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST1B (scalar plus vector)

Scatter store bytes from a vector (vector index)

Scatter store of bytes from the active elements of a vector register to the memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally sign or zero-extended from 32 to 64 bits. Inactive elements are not written to memory.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 3 classes: 32-bit unpacked unscaled offset , 32-bit unscaled offset and 64-bit unscaled offset

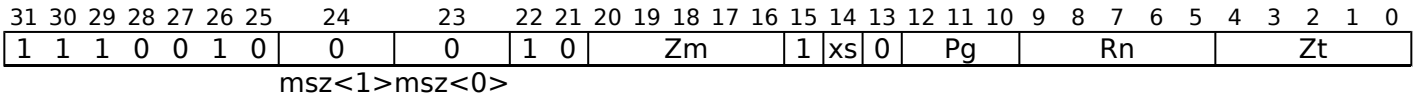
32-bit unpacked unscaled offset



ST1B { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

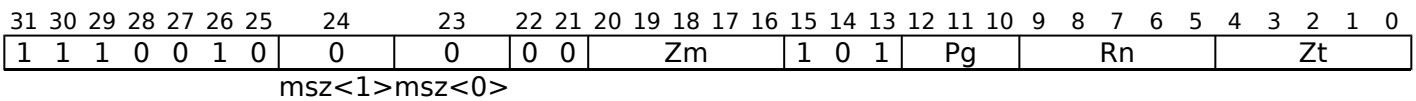
32-bit unscaled offset



ST1B { <Zt>.S }, <Pg>, [<Xn|SP>, <Zm>.S, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

64-bit unscaled offset



ST1B { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST1B (vector plus immediate)

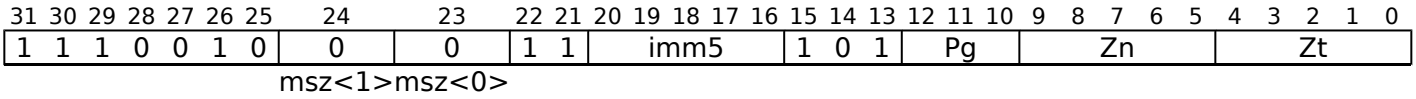
Scatter store bytes from a vector (immediate index)

Scatter store of bytes from the active elements of a vector register to the memory addresses generated by a vector base plus immediate index. The index is in the range 0 to 31. Inactive elements are not written to memory.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

It has encodings from 2 classes: 32-bit element and 64-bit element

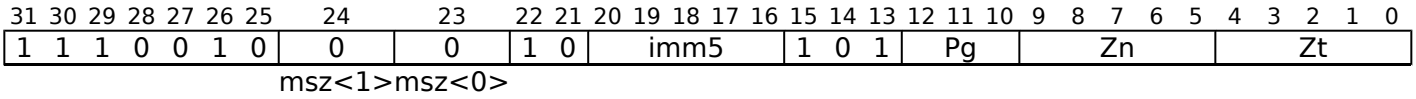
32-bit element



ST1B { <Zt>.S }, <Pg>, [<Zn>.S{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
integer offset = UInt(imm5);
```

64-bit element



ST1B { <Zt>.D }, <Pg>, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
integer offset = UInt(imm5);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm>	Is the optional unsigned immediate byte offset, in the range 0 to 31, defaulting to 0, encoded in the "imm5" field.

Operation

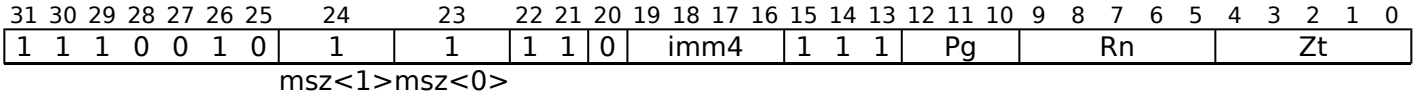
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) src;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST1D (scalar plus immediate)

Contiguous store doublewords from vector (immediate index)

Contiguous store of doublewords from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.



ST1D { <Zt>.D }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

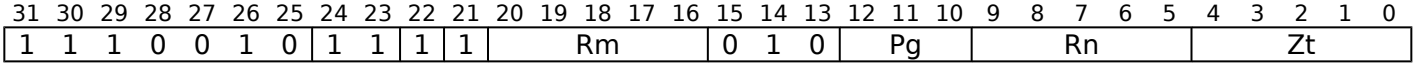
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) src;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST1D (scalar plus scalar)

Contiguous store doublewords from vector (scalar index)

Contiguous store of doublewords from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.



ST1D { <Zt>.D }, <Pg>, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
```

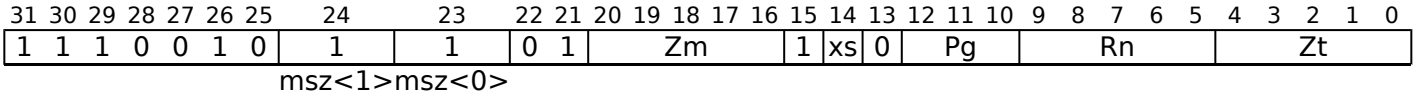
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST1D (scalar plus vector)

Scatter store doublewords from a vector (vector index)

Scatter store of doublewords from the active elements of a vector register to the memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 8. Inactive elements are not written to memory.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 4 classes: 32-bit unpacked scaled offset , 32-bit unpacked unscaled offset , 64-bit scaled offset and 64-bit unscaled offset

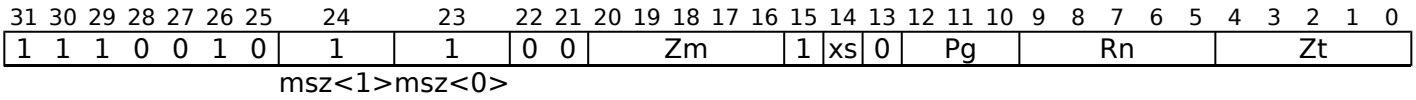
32-bit unpacked scaled offset



ST1D { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #3]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 3;
```

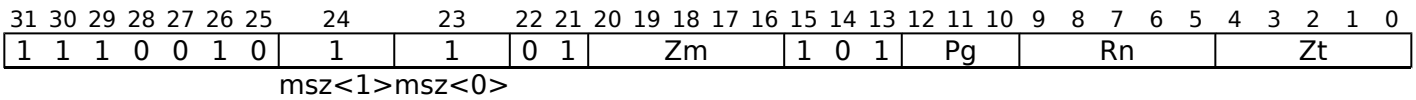
32-bit unpacked unscaled offset



ST1D { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

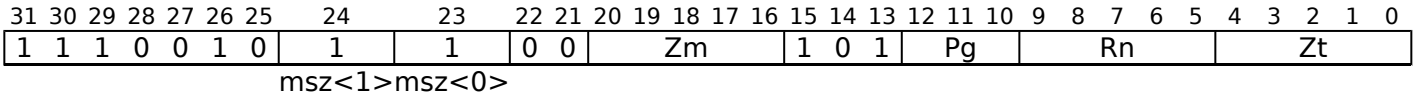
64-bit scaled offset



ST1D { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, LSL #3]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 3;
```

64-bit unscaled offset



ST1D { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Zm>	Is the name of the offset scalable vector register, encoded in the "Zm" field.
<mod>	Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

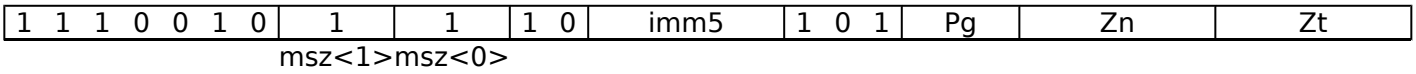
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST1D (vector plus immediate)

Scatter store doublewords from a vector (immediate index)

Scatter store of doublewords from the active elements of a vector register to the memory addresses generated by a vector base plus immediate index. The index is a multiple of 8 in the range 0 to 248. Inactive elements are not written to memory.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.



ST1D { <Zt>.D }, <Pg>, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
integer offset = UInt(imm5);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm>	Is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 248, defaulting to 0, encoded in the "imm5" field.

Operation

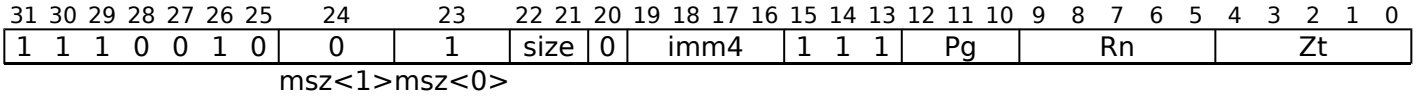
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) src;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST1H (scalar plus immediate)

Contiguous store halfwords from vector (immediate index)

Contiguous store of halfwords from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.



ST1H { <Zt>.<T> }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8 << UInt(size);
constant integer msize = 16;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

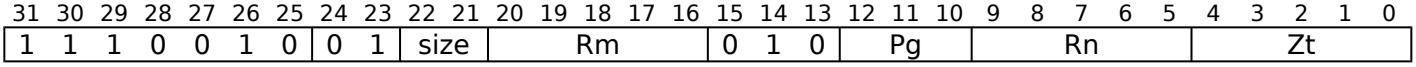
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) src;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST1H (scalar plus scalar)

Contiguous store halfwords from vector (scalar index)

Contiguous store of halfwords from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.



ST1H { <Zt>.<T> }, <Pg>, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8 << UInt(size);
constant integer msize = 16;
```

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
```

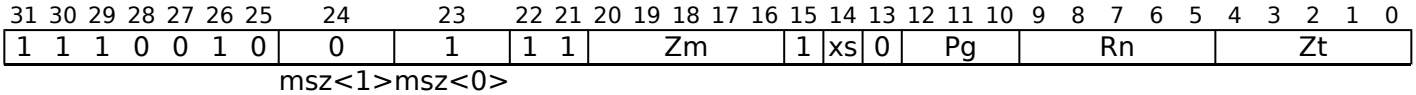
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST1H (scalar plus vector)

Scatter store halfwords from a vector (vector index)

Scatter store of halfwords from the active elements of a vector register to the memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 2. Inactive elements are not written to memory.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 6 classes: 32-bit scaled offset , 32-bit unpacked scaled offset , 32-bit unpacked unscaled offset , 32-bit unscaled offset , 64-bit scaled offset and 64-bit unscaled offset

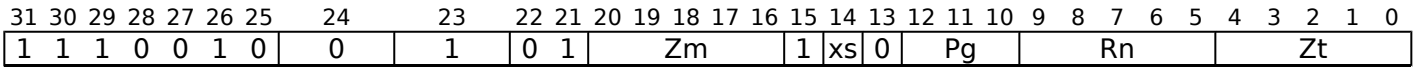
32-bit scaled offset



ST1H { <Zt>.S }, <Pg>, [<Xn|SP>, <Zm>.S, <mod> #1]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```

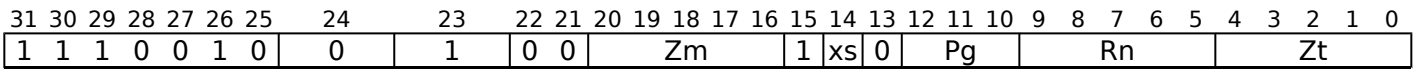
32-bit unpacked scaled offset



ST1H { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #1]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 1;
```

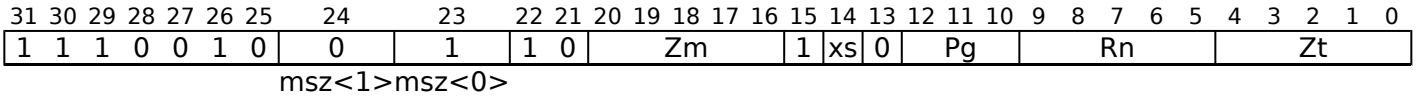
32-bit unpacked unscaled offset



ST1H { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

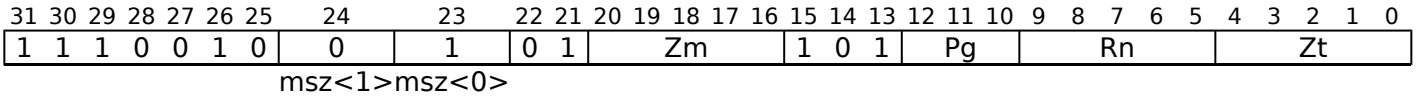
32-bit unscaled offset



ST1H { <Zt>.S }, <Pg>, [<Xn|SP>, <Zm>.S, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

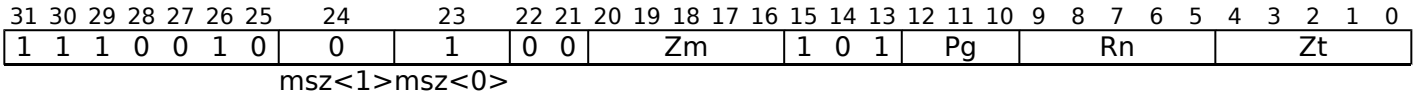
64-bit scaled offset



ST1H { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, LSL #1]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 1;
```

64-bit unscaled offset



ST1H { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
```

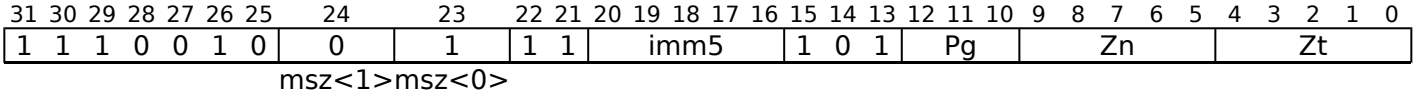
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST1H (vector plus immediate)

Scatter store halfwords from a vector (immediate index)

Scatter store of halfwords from the active elements of a vector register to the memory addresses generated by a vector base plus immediate index. The index is a multiple of 2 in the range 0 to 62. Inactive elements are not written to memory.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 2 classes: 32-bit element and 64-bit element

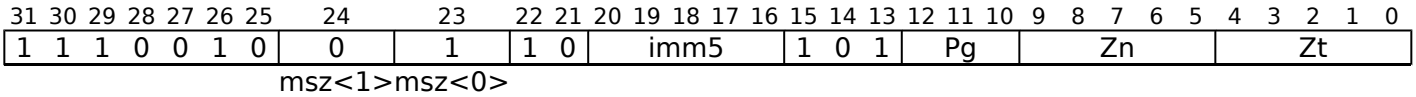
32-bit element



ST1H { <Zt>.S }, <Pg>, [<Zn>.S{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
integer offset = UInt(imm5);
```

64-bit element



ST1H { <Zt>.D }, <Pg>, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
integer offset = UInt(imm5);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm>	Is the optional unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0, encoded in the "imm5" field.

Operation

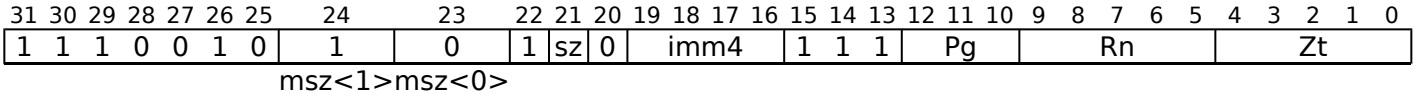
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) src;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST1W (scalar plus immediate)

Contiguous store words from vector (immediate index)

Contiguous store of words from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.



ST1W { <Zt>.<T> }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32 << UInt(sz);
constant integer msize = 32;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<T> Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

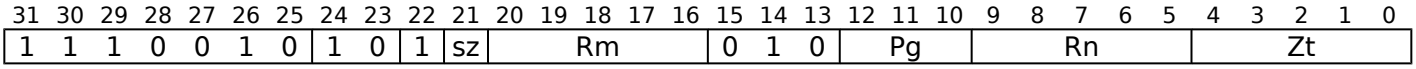
Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) src;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
```

ST1W (scalar plus scalar)

Contiguous store words from vector (scalar index)

Contiguous store of words from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.



ST1W { <Zt>.<T> }, <Pg>, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32 << UInt(sz);
constant integer msize = 32;
```

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<T> Is the size specifier, encoded in “sz”:

sz	<T>
0	S
1	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

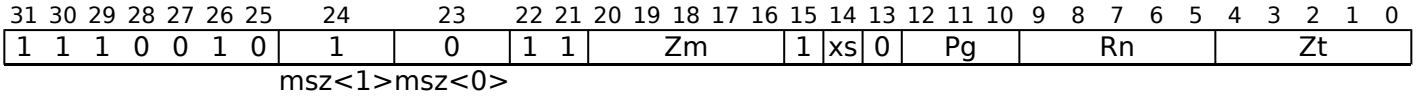
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
```

ST1W (scalar plus vector)

Scatter store words from a vector (vector index)

Scatter store of words from the active elements of a vector register to the memory addresses generated by a 64-bit scalar base plus vector index. The index values are optionally first sign or zero-extended from 32 to 64 bits and then optionally multiplied by 4. Inactive elements are not written to memory.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 6 classes: 32-bit scaled offset , 32-bit unpacked scaled offset , 32-bit unpacked unscaled offset , 32-bit unscaled offset , 64-bit scaled offset and 64-bit unscaled offset

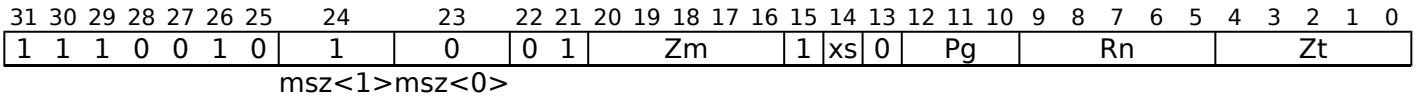
32-bit scaled offset



ST1W { <Zt>.S }, <Pg>, [<Xn|SP>, <Zm>.S, <mod> #2]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 2;
```

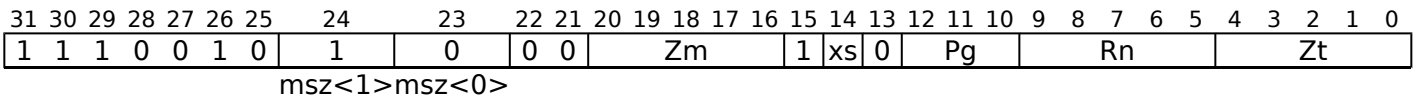
32-bit unpacked scaled offset



ST1W { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod> #2]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 2;
```

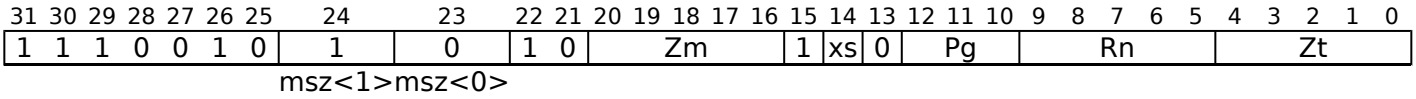
32-bit unpacked unscaled offset



ST1W { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

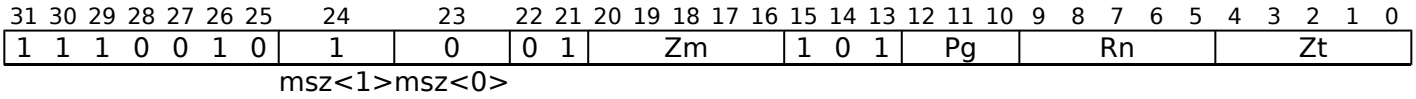
32-bit unscaled offset



ST1W { <Zt>.S }, <Pg>, [<Xn|SP>, <Zm>.S, <mod>]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
integer offs_size = 32;
boolean offs_unsigned = xs == '0';
integer scale = 0;
```

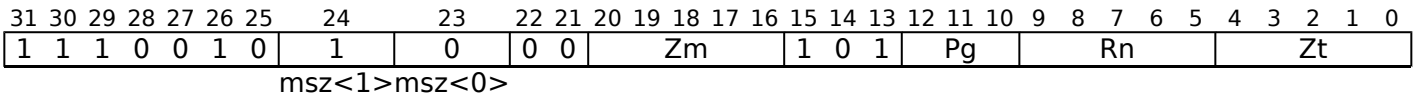
64-bit scaled offset



ST1W { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D, LSL #2]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 2;
```

64-bit unscaled offset



ST1W { <Zt>.D }, <Pg>, [<Xn|SP>, <Zm>.D]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Zm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offs_size = 64;
boolean offs_unsigned = TRUE;
integer scale = 0;
```

Assembler Symbols

<Zt> Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Zm> Is the name of the offset scalable vector register, encoded in the "Zm" field.

<mod> Is the index extend and shift specifier, encoded in “xs”:

xs	<mod>
0	UXTW
1	SXTW

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(VL) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = Z[m, VL];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer off = Int(Elem[offset, e, esize]<offs_size-1:0>, offs_unsigned);
        bits(64) addr = base + (off << scale);
        Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
```

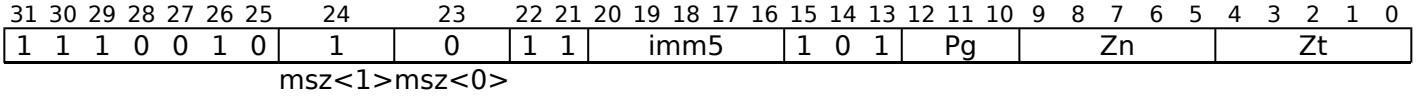
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST1W (vector plus immediate)

Scatter store words from a vector (immediate index)

Scatter store of words from the active elements of a vector register to the memory addresses generated by a vector base plus immediate index. The index is a multiple of 4 in the range 0 to 124. Inactive elements are not written to memory.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 2 classes: 32-bit element and 64-bit element

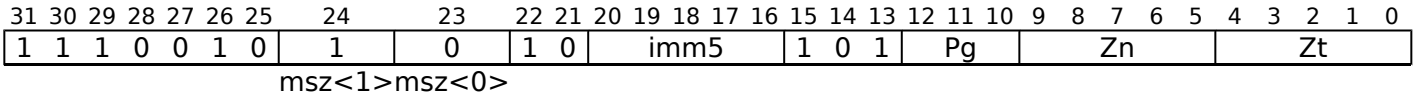
32-bit element



ST1W { <Zt>.S }, <Pg>, [<Zn>.S{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
integer offset = UInt(imm5);
```

64-bit element



ST1W { <Zt>.D }, <Pg>, [<Zn>.D{, #<imm>}]

```
if !HaveSVE() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
integer offset = UInt(imm5);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the base scalable vector register, encoded in the "Zn" field.
<imm>	Is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0, encoded in the "imm5" field.

Operation

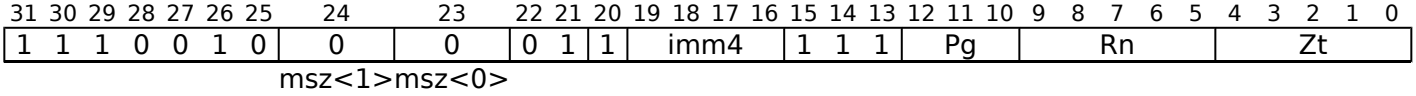
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(VL) src;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset * mbytes;
        Mem[addr, mbytes, AccType_SVE] = Elem[src, e, esize]<msize-1:0>;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST2B (scalar plus immediate)

Contiguous store two-byte structures from two vectors (immediate index)

Contiguous store two-byte structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication,
Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.



ST2B { <Zt1>.B, <Zt2>.B }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
integer offset = SInt(imm4);
constant integer nreg = 2;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

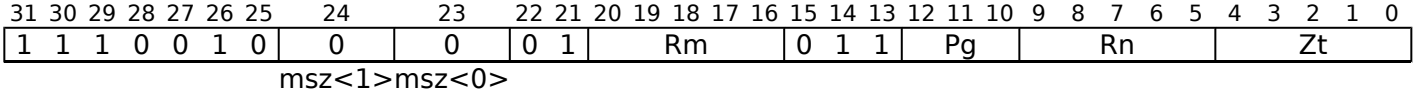
Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```

ST2B (scalar plus scalar)

Contiguous store two-byte structures from two vectors (scalar index)

Contiguous store two-byte structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.
Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.



ST2B { <Zt1>.B, <Zt2>.B }, <Pg>, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
constant integer nreg = 2;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

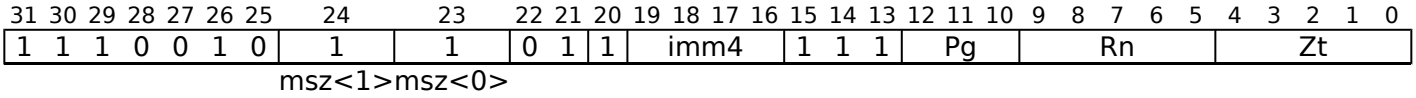
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST2D (scalar plus immediate)

Contiguous store two-doubleword structures from two vectors (immediate index)

Contiguous store two-doubleword structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication,
Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.



ST2D { <Zt1>.D, <Zt2>.D }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
integer offset = SInt(imm4);
constant integer nreg = 2;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

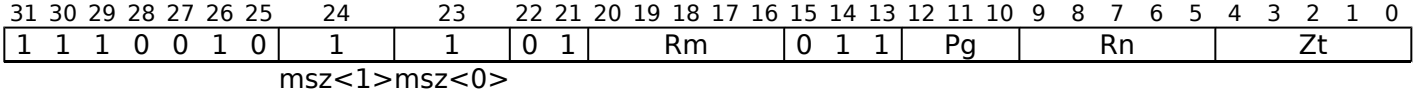
Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```

ST2D (scalar plus scalar)

Contiguous store two-doubleword structures from two vectors (scalar index)

Contiguous store two-doubleword structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.
Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.



ST2D { <Zt1>.D, <Zt2>.D }, <Pg>, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer nreg = 2;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

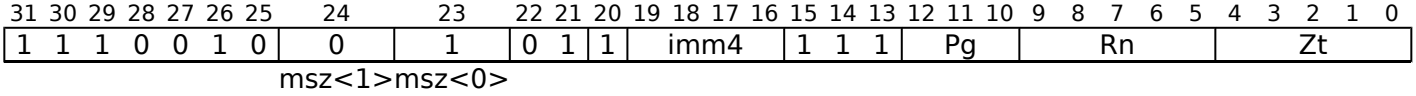
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST2H (scalar plus immediate)

Contiguous store two-halfword structures from two vectors (immediate index)

Contiguous store two-halfword structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication,
Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive halfwords in memory which make up each structure. Inactive structures are not written to memory.



ST2H { <Zt1>.H, <Zt2>.H }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
integer offset = SInt(imm4);
constant integer nreg = 2;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

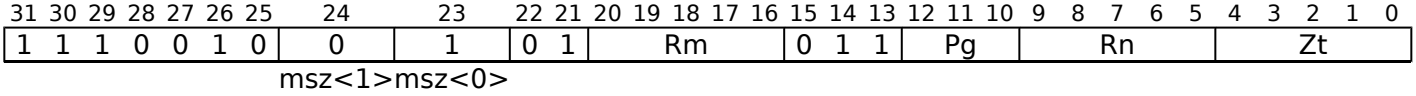
Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```

ST2H (scalar plus scalar)

Contiguous store two-halfword structures from two vectors (scalar index)

Contiguous store two-halfword structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.
Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive halfwords in memory which make up each structure. Inactive structures are not written to memory.



ST2H { <Zt1>.H, <Zt2>.H }, <Pg>, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer nreg = 2;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

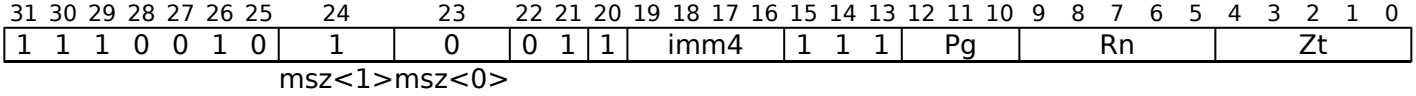
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST2W (scalar plus immediate)

Contiguous store two-word structures from two vectors (immediate index)

Contiguous store two-word structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 2 in the range -16 to 14 that is multiplied by the vector's in-memory size, irrespective of predication,
Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive words in memory which make up each structure. Inactive structures are not written to memory.



ST2W { <Zt1>.S, <Zt2>.S }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
integer offset = SInt(imm4);
constant integer nreg = 2;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, a multiple of 2 in the range -16 to 14, defaulting to 0, encoded in the "imm4" field.

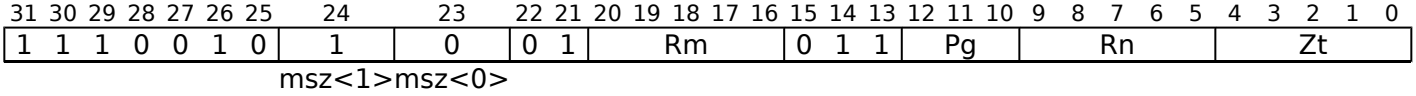
Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```

ST2W (scalar plus scalar)

Contiguous store two-word structures from two vectors (scalar index)

Contiguous store two-word structures, each from the same element number in two vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by two. The index register is not updated by the instruction.
Each predicate element applies to the same element number in each of the two vector registers, or equivalently to the two consecutive words in memory which make up each structure. Inactive structures are not written to memory.



ST2W { <Zt1>.S, <Zt2>.S }, <Pg>, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer nreg = 2;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

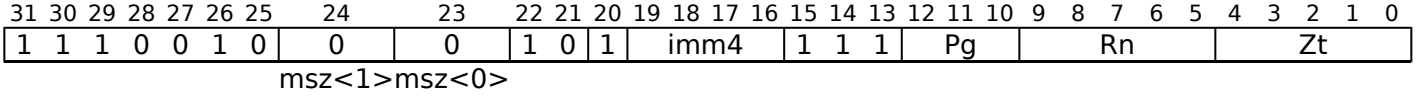
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..1] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST3B (scalar plus immediate)

Contiguous store three-byte structures from three vectors (immediate index)

Contiguous store three-byte structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication,
Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.



ST3B { <Zt1>.B, <Zt2>.B, <Zt3>.B }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
integer offset = SInt(imm4);
constant integer nreg = 3;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

Operation

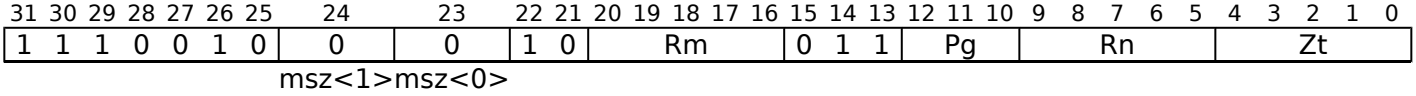
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST3B (scalar plus scalar)

Contiguous store three-byte structures from three vectors (scalar index)

Contiguous store three-byte structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.
Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.



ST3B { <Zt1>.B, <Zt2>.B, <Zt3>.B }, <Pg>, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
constant integer nreg = 3;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

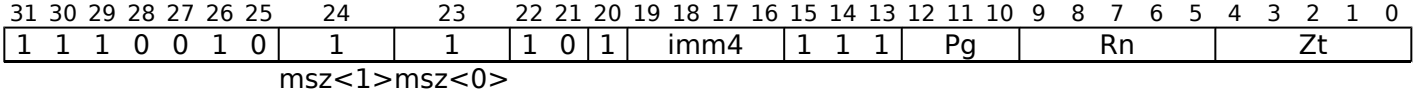
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST3D (scalar plus immediate)

Contiguous store three-doubleword structures from three vectors (immediate index)

Contiguous store three-doubleword structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication,
Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.



ST3D { <Zt1>.D, <Zt2>.D, <Zt3>.D }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
integer offset = SInt(imm4);
constant integer nreg = 3;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

Operation

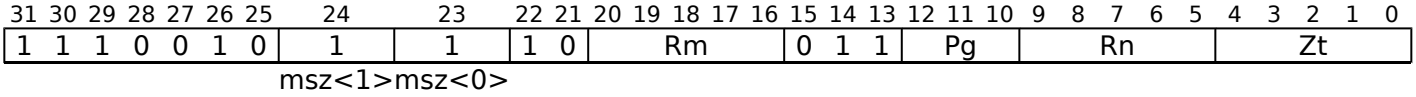
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST3D (scalar plus scalar)

Contiguous store three-doubleword structures from three vectors (scalar index)

Contiguous store three-doubleword structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.
Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.



ST3D { <Zt1>.D, <Zt2>.D, <Zt3>.D }, <Pg>, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer nreg = 3;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

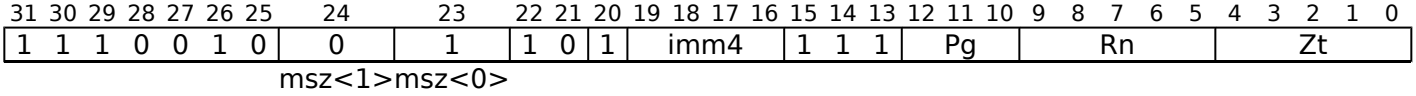
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST3H (scalar plus immediate)

Contiguous store three-halfword structures from three vectors (immediate index)

Contiguous store three-halfword structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication,
Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive halfwords in memory which make up each structure. Inactive structures are not written to memory.



ST3H { <Zt1>.H, <Zt2>.H, <Zt3>.H }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
integer offset = SInt(imm4);
constant integer nreg = 3;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

Operation

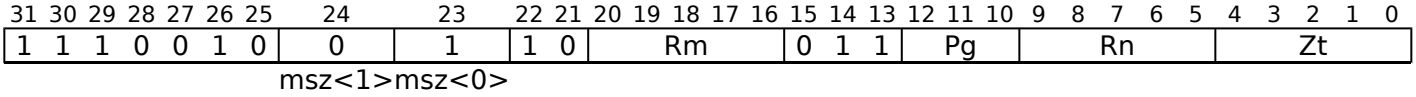
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST3H (scalar plus scalar)

Contiguous store three-halfword structures from three vectors (scalar index)

Contiguous store three-halfword structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.
Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive halfwords in memory which make up each structure. Inactive structures are not written to memory.



ST3H { <Zt1>.H, <Zt2>.H, <Zt3>.H }, <Pg>, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer nreg = 3;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

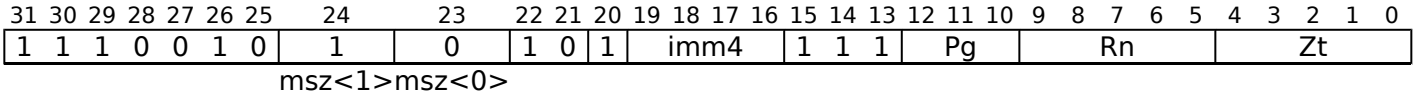
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST3W (scalar plus immediate)

Contiguous store three-word structures from three vectors (immediate index)

Contiguous store three-word structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 3 in the range -24 to 21 that is multiplied by the vector's in-memory size, irrespective of predication,
Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive words in memory which make up each structure. Inactive structures are not written to memory.



ST3W { <Zt1>.S, <Zt2>.S, <Zt3>.S }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
integer offset = SInt(imm4);
constant integer nreg = 3;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, a multiple of 3 in the range -24 to 21, defaulting to 0, encoded in the "imm4" field.

Operation

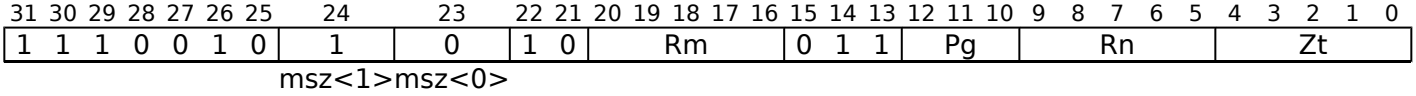
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST3W (scalar plus scalar)

Contiguous store three-word structures from three vectors (scalar index)

Contiguous store three-word structures, each from the same element number in three vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by three. The index register is not updated by the instruction.
Each predicate element applies to the same element number in each of the three vector registers, or equivalently to the three consecutive words in memory which make up each structure. Inactive structures are not written to memory.



ST3W { <Zt1>.S, <Zt2>.S, <Zt3>.S }, <Pg>, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer nreg = 3;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

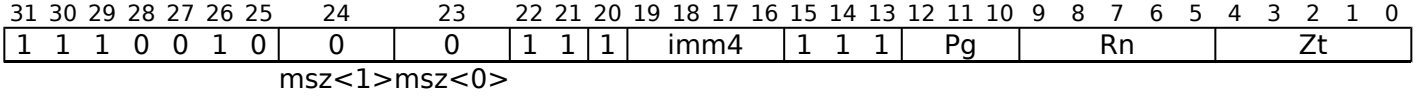
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..2] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST4B (scalar plus immediate)

Contiguous store four-byte structures from four vectors (immediate index)

Contiguous store four-byte structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication,
Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.



ST4B { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
integer offset = SInt(imm4);
constant integer nreg = 4;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4>	Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

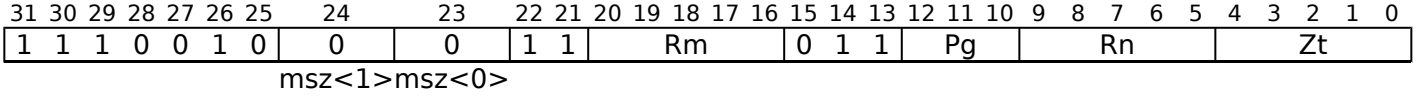
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST4B (scalar plus scalar)

Contiguous store four-byte structures from four vectors (scalar index)

Contiguous store four-byte structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.
Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive bytes in memory which make up each structure. Inactive structures are not written to memory.



ST4B { <Zt1>.B, <Zt2>.B, <Zt3>.B, <Zt4>.B }, <Pg>, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
constant integer nreg = 4;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4>	Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

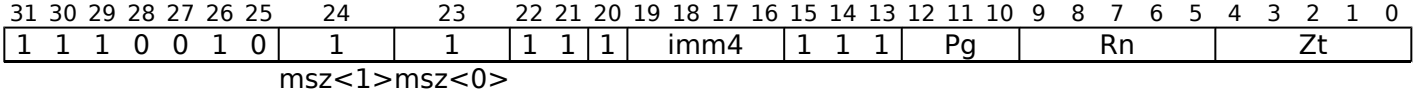
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST4D (scalar plus immediate)

Contiguous store four-doubleword structures from four vectors (immediate index)

Contiguous store four-doubleword structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication,
Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.



ST4D { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
integer offset = SInt(imm4);
constant integer nreg = 4;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4>	Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

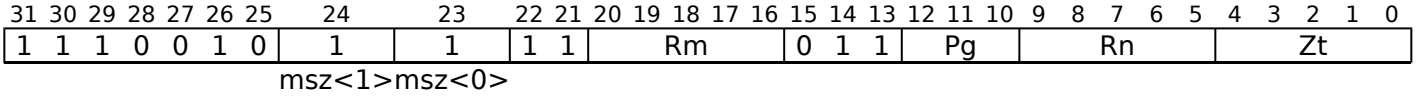
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST4D (scalar plus scalar)

Contiguous store four-doubleword structures from four vectors (scalar index)

Contiguous store four-doubleword structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.
Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive doublewords in memory which make up each structure. Inactive structures are not written to memory.



ST4D { <Zt1>.D, <Zt2>.D, <Zt3>.D, <Zt4>.D }, <Pg>, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer nreg = 4;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4>	Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

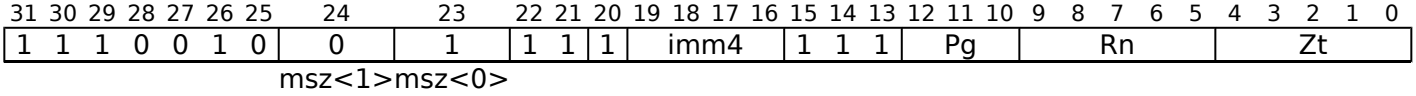
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST4H (scalar plus immediate)

Contiguous store four-halfword structures from four vectors (immediate index)

Contiguous store four-halfword structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication,
Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive halfwords in memory which make up each structure. Inactive structures are not written to memory.



ST4H { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
integer offset = SInt(imm4);
constant integer nreg = 4;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4>	Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

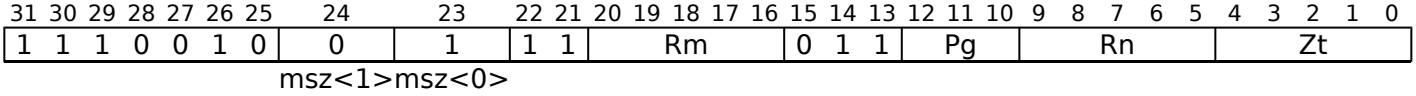
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST4H (scalar plus scalar)

Contiguous store four-halfword structures from four vectors (scalar index)

Contiguous store four-halfword structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.
Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive halfwords in memory which make up each structure. Inactive structures are not written to memory.



ST4H { <Zt1>.H, <Zt2>.H, <Zt3>.H, <Zt4>.H }, <Pg>, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
constant integer nreg = 4;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4>	Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

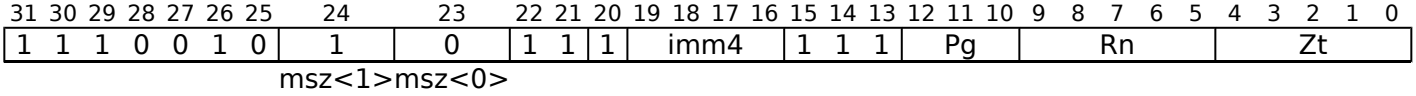
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST4W (scalar plus immediate)

Contiguous store four-word structures from four vectors (immediate index)

Contiguous store four-word structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and an immediate index which is a multiple of 4 in the range -32 to 28 that is multiplied by the vector's in-memory size, irrespective of predication,
Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive words in memory which make up each structure. Inactive structures are not written to memory.



ST4W { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
integer offset = SInt(imm4);
constant integer nreg = 4;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4>	Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, a multiple of 4 in the range -32 to 28, defaulting to 0, encoded in the "imm4" field.

Operation

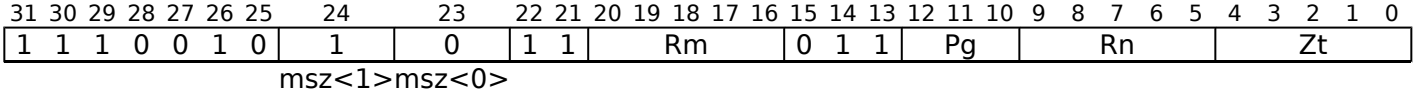
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = (offset * elements * nreg) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST4W (scalar plus scalar)

Contiguous store four-word structures from four vectors (scalar index)

Contiguous store four-word structures, each from the same element number in four vector registers to the memory address generated by a 64-bit scalar base and a 64-bit scalar index register scaled by the element size (LSL option) and added to the base address. After each structure access the index value is incremented by four. The index register is not updated by the instruction.
Each predicate element applies to the same element number in each of the four vector registers, or equivalently to the four consecutive words in memory which make up each structure. Inactive structures are not written to memory.



ST4W { <Zt1>.S, <Zt2>.S, <Zt3>.S, <Zt4>.S }, <Pg>, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer nreg = 4;
```

Assembler Symbols

<Zt1>	Is the name of the first scalable vector register to be transferred, encoded in the "Zt" field.
<Zt2>	Is the name of the second scalable vector register to be transferred, encoded as "Zt" plus 1 modulo 32.
<Zt3>	Is the name of the third scalable vector register to be transferred, encoded as "Zt" plus 2 modulo 32.
<Zt4>	Is the name of the fourth scalable vector register to be transferred, encoded as "Zt" plus 3 modulo 32.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

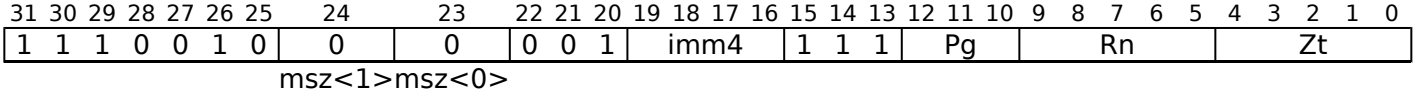
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(PL) mask = P[g, PL];
bits(64) offset;
constant integer mbytes = esize DIV 8;
array [0..3] of bits(VL) values;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];

for r = 0 to nreg-1
    values[r] = Z[(t+r) MOD 32, VL];

for e = 0 to elements-1
    for r = 0 to nreg-1
        if ElemP[mask, e, esize] == '1' then
            integer eoff = UInt(offset) + (e * nreg) + r;
            bits(64) addr = base + eoff * mbytes;
            Mem[addr, mbytes, AccType_SVE] = Elem[values[r], e, esize];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STNT1B (scalar plus immediate)

Contiguous store non-temporal bytes from vector (immediate index)

Contiguous store non-temporal of bytes from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.
A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.



STNT1B { <Zt>.B }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 8;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

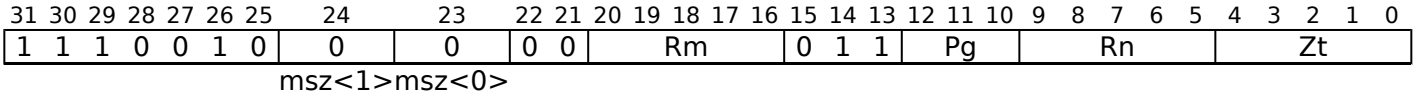
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
constant integer mbytes = esize DIV 8;
bits(VL) src;
bits(PL) mask = P[g, PL];

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Mem[addr, mbytes, AccType_SVESTREAM] = Elem[src, e, esize];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STNT1B (scalar plus scalar)

Contiguous store non-temporal bytes from vector (scalar index)

Contiguous store non-temporal of bytes from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.
A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.



STNT1B { <Zt>.B }, <Pg>, [<Xn|SP>, <Xm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 8;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(64) offset;
bits(VL) src;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        Mem[addr, mbytes, AccType_SVESTREAM] = Elem[src, e, esize];
```

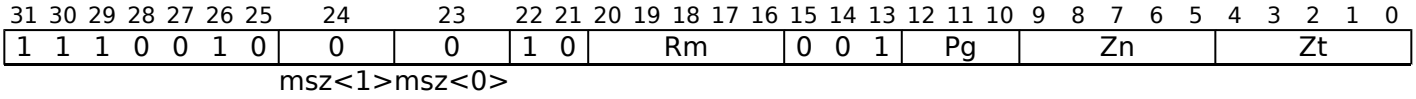
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STNT1B (vector plus scalar)

Scatter store non-temporal bytes

Scatter store non-temporal of bytes from the active elements of a vector register to the memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements are not written to memory.
A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 2 classes: 32-bit unscaled offset and 64-bit unscaled offset

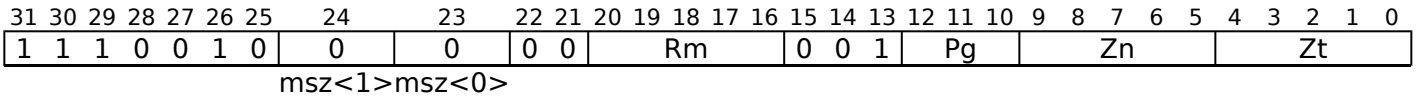
32-bit unscaled offset



STNT1B { <Zt>.S }, <Pg>, [<Zn>.S{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 8;
```

64-bit unscaled offset



STNT1B { <Zt>.D }, <Pg>, [<Zn>.D{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 8;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the base scalable vector register, encoded in the "Zn" field.
<Xm>	Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

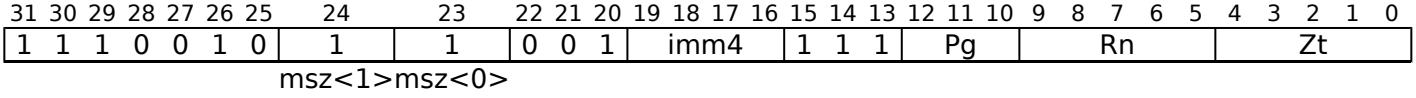
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    offset = X[m, 64];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
        Mem[addr, mbytes, AccType_SVESTREAM] = Elem[src, e, esize]<msize-1:0>;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STNT1D (scalar plus immediate)

Contiguous store non-temporal doublewords from vector (immediate index)

Contiguous store non-temporal of doublewords from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.
A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.



STNT1D { <Zt>.D }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 64;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
constant integer mbytes = esize DIV 8;
bits(VL) src;
bits(PL) mask = P[g, PL];

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Mem[addr, mbytes, AccType_SVESTREAM] = Elem[src, e, esize];
```

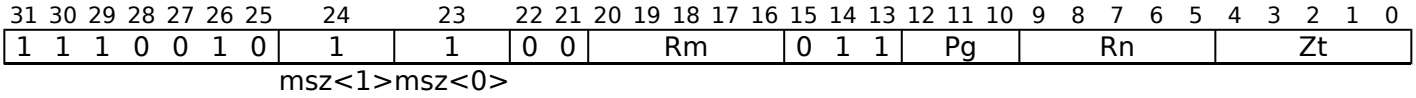
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STNT1D (scalar plus scalar)

Contiguous store non-temporal doublewords from vector (scalar index)

Contiguous store non-temporal of doublewords from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 8 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.

A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.



STNT1D { <Zt>.D }, <Pg>, [<Xn|SP>, <Xm>, LSL #3]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

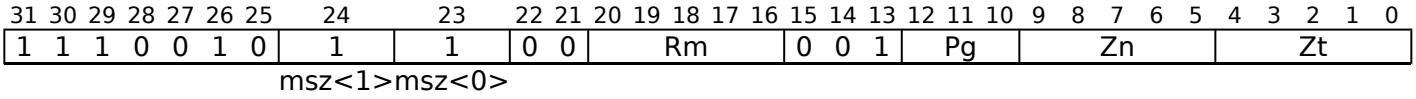
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(64) offset;
bits(VL) src;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        Mem[addr, mbytes, AccType_SVESTREAM] = Elem[src, e, esize];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STNT1D (vector plus scalar)

Scatter store non-temporal doublewords

Scatter store non-temporal of doublewords from the active elements of a vector register to the memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements are not written to memory.
A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.



STNT1D { <Zt>.D }, <Pg>, [<Zn>.D{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 64;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the base scalable vector register, encoded in the "Zn" field.
<Xm>	Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

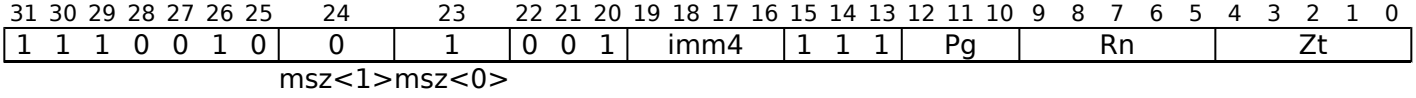
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    offset = X[m, 64];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
        Mem[addr, mbytes, AccType_SVESTREAM] = Elem[src, e, esize]<msize-1:0>;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STNT1H (scalar plus immediate)

Contiguous store non-temporal halfwords from vector (immediate index)

Contiguous store non-temporal of halfwords from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.
A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.



STNT1H { <Zt>.H }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 16;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

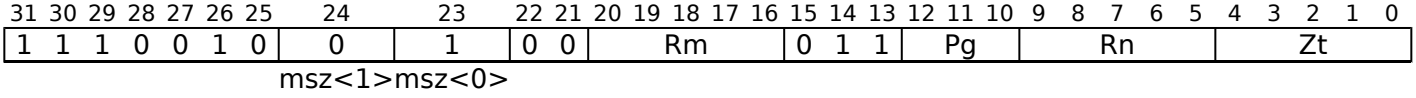
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
constant integer mbytes = esize DIV 8;
bits(VL) src;
bits(PL) mask = P[g, PL];

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Mem[addr, mbytes, AccType_SVESTREAM] = Elem[src, e, esize];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STNT1H (scalar plus scalar)

Contiguous store non-temporal halfwords from vector (scalar index)

Contiguous store non-temporal of halfwords from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 2 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory.
A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.



STNT1H { <Zt>.H }, <Pg>, [<Xn|SP>, <Xm>, LSL #1]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 16;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(64) offset;
bits(VL) src;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        Mem[addr, mbytes, AccType_SVESTREAM] = Elem[src, e, esize];
```

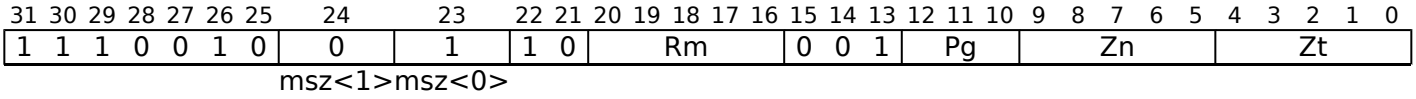
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STNT1H (vector plus scalar)

Scatter store non-temporal halfwords

Scatter store non-temporal of halfwords from the active elements of a vector register to the memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements are not written to memory.
A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 2 classes: 32-bit unscaled offset and 64-bit unscaled offset

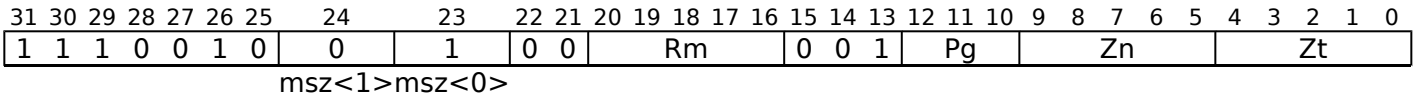
32-bit unscaled offset



STNT1H { <Zt>.S }, <Pg>, [<Zn>.S{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 16;
```

64-bit unscaled offset



STNT1H { <Zt>.D }, <Pg>, [<Zn>.D{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 16;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the base scalable vector register, encoded in the "Zn" field.
<Xm>	Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

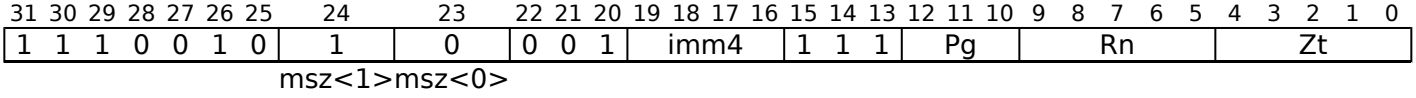
```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    offset = X[m, 64];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
        Mem[addr, mbytes, AccType_SVESTREAM] = Elem[src, e, esize]<msize-1:0>;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STNT1W (scalar plus immediate)

Contiguous store non-temporal words from vector (immediate index)

Contiguous store non-temporal of words from elements of a vector register to the memory address generated by a 64-bit scalar base and immediate index in the range -8 to 7 which is multiplied by the vector's in-memory size, irrespective of predication, and added to the base address. Inactive elements are not written to memory.
A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.



STNT1W { <Zt>.S }, <Pg>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer g = UInt(Pg);
constant integer esize = 32;
integer offset = SInt(imm4);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -8 to 7, defaulting to 0, encoded in the "imm4" field.

Operation

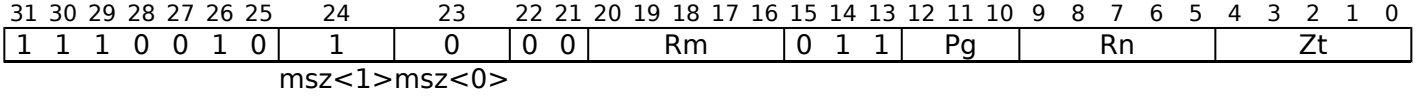
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
constant integer mbytes = esize DIV 8;
bits(VL) src;
bits(PL) mask = P[g, PL];

if HaveMTEExt() then SetTagCheckedInstruction(n != 31);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer eoff = (offset * elements) + e;
        bits(64) addr = base + eoff * mbytes;
        Mem[addr, mbytes, AccType_SVESTREAM] = Elem[src, e, esize];
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STNT1W (scalar plus scalar)

Contiguous store non-temporal words from vector (scalar index)

Contiguous store non-temporal of words from elements of a vector register to the memory address generated by a 64-bit scalar base and scalar index which is multiplied by 4 and added to the base address. After each element access the index value is incremented, but the index register is not updated. Inactive elements are not written to memory. A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.



STNT1W { <Zt>.S }, <Pg>, [<Xn|SP>, <Xm>, LSL #2]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if Rm == '11111' then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<Xm>	Is the 64-bit name of the general-purpose offset register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(64) base;
bits(64) offset;
bits(VL) src;
bits(PL) mask = P[g, PL];
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if !AnyActiveElement(mask, esize) then
    if n == 31 && ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
else
    if n == 31 then CheckSPAlignment();
    base = if n == 31 then SP[] else X[n, 64];
    offset = X[m, 64];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = base + (UInt(offset) + e) * mbytes;
        Mem[addr, mbytes, AccType_SVESTREAM] = Elem[src, e, esize];
```

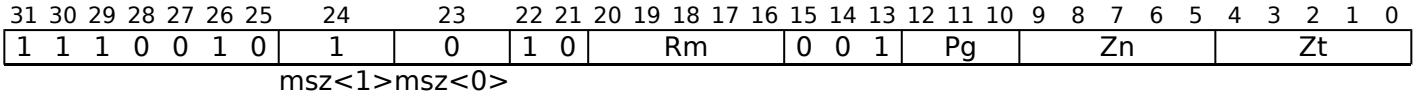
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STNT1W (vector plus scalar)

Scatter store non-temporal words

Scatter store non-temporal of words from the active elements of a vector register to the memory addresses generated by a vector base plus a 64-bit unscaled scalar register offset. Inactive elements are not written to memory.
A non-temporal store is a hint to the system that this data is unlikely to be referenced again soon.
This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.
It has encodings from 2 classes: 32-bit unscaled offset and 64-bit unscaled offset

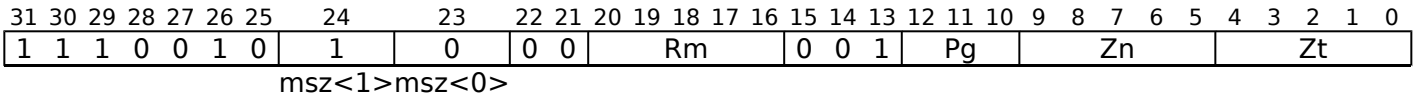
32-bit unscaled offset



STNT1W { <Zt>.S }, <Pg>, [<Zn>.S{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 32;
constant integer msize = 32;
```

64-bit unscaled offset



STNT1W { <Zt>.D }, <Pg>, [<Zn>.D{, <Xm>}]

```
if !HaveSVE2() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Zn);
integer m = UInt(Rm);
integer g = UInt(Pg);
constant integer esize = 64;
constant integer msize = 32;
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the base scalable vector register, encoded in the "Zn" field.
<Xm>	Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) base;
bits(64) offset;
bits(VL) src;
constant integer mbytes = msize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if AnyActiveElement(mask, esize) then
    base = Z[n, VL];
    offset = X[m, 64];
    src = Z[t, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(64) addr = ZeroExtend(Elem[base, e, esize], 64) + offset;
        Mem[addr, mbytes, AccType_SVESTREAM] = Elem[src, e, esize]<msize-1:0>;
```

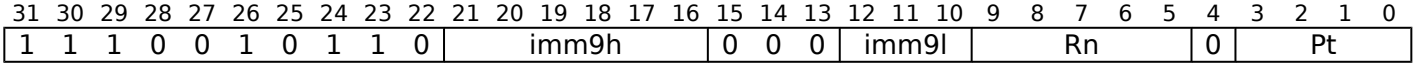
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STR (predicate)

Store predicate register

Store a predicate register to a memory address generated by a 64-bit scalar base, plus an immediate offset in the range -256 to 255 which is multiplied by the current predicate register size in bytes. This instruction is unpredicated.

The store is performed as contiguous byte accesses, each containing 8 consecutive predicate bits in ascending element order, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then a general-purpose base register must be aligned to 2 bytes.



STR <Pt>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Pt);
integer n = UInt(Rn);
integer imm = SInt(imm9h:imm9l);
```

Assembler Symbols

<Pt>	Is the name of the scalable predicate transfer register, encoded in the "Pt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9h:imm9l" fields.

Operation

```
CheckSVEEnabled();
constant integer PL = CurrentVL DIV 8;
constant integer elements = PL DIV 8;
bits(PL) src;
bits(64) base;
integer offset = imm * elements;

if n == 31 then
    CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n, 64];

src = P[t, PL];
boolean aligned = AArch64.CheckAlignment(base + offset, 2, AccType_SVE, TRUE);
for e = 0 to elements-1
    AArch64.MemSingle[base + offset, 1, AccType_SVE, aligned] = Elem[src, e, 8];
    offset = offset + 1;
```

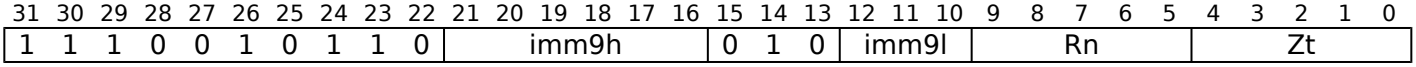
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STR (vector)

Store vector register

Store a vector register to a memory address generated by a 64-bit scalar base, plus an immediate offset in the range -256 to 255 which is multiplied by the current vector register size in bytes. This instruction is unpredicated.

The store is performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then the base register must be aligned to 16 bytes.



STR <Zt>, [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
integer t = UInt(Zt);
integer n = UInt(Rn);
integer imm = SInt(imm9h:imm9l);
```

Assembler Symbols

<Zt>	Is the name of the scalable vector register to be transferred, encoded in the "Zt" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.
<imm>	Is the optional signed immediate vector offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9h:imm9l" fields.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV 8;
bits(VL) src;
bits(64) base;
integer offset = imm * elements;

if n == 31 then
    CheckSPAlignment();
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n, 64];

src = Z[t, VL];
boolean aligned = AArch64.CheckAlignment(base + offset, 16, AccType_SVE, TRUE);
for e = 0 to elements-1
    AArch64.MemSingle[base + offset, 1, AccType_SVE, aligned] = Elem[src, e, 8];
    offset = offset + 1;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

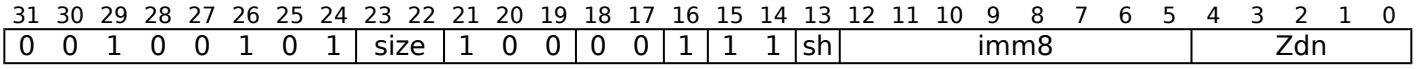
SUB (immediate)

Subtract immediate (unpredicated)

Subtract an unsigned immediate from each element of the source vector, and destructively place the results in the corresponding elements of the source vector. This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<uimm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".



SUB <Zdn>.<T>, <Zdn>.<T>, #<imm>{, <shift>}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = UInt(imm8);
if sh == '1' then imm = imm << 8;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<imm> Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #8

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    Elem[result, e, esize] = element1 - imm;

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.

    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

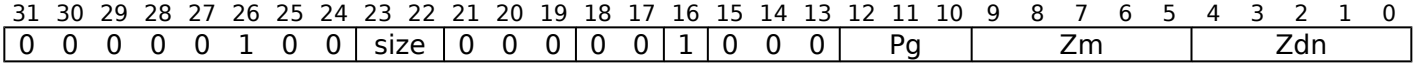
- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SUB (vectors, predicated)

Subtract vectors (predicated)

Subtract active elements of the second source vector from corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



SUB <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = element1 - element2;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

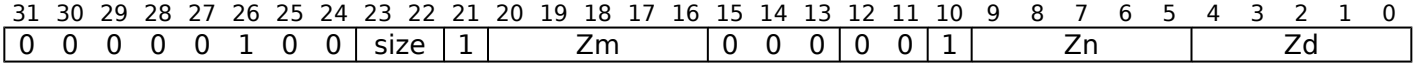
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SUB (vectors, unpredicated)

Subtract vectors (unpredicated)

Subtract all elements of the second source vector from corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. This instruction is unpredicated.



SUB <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = element1 - element2;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

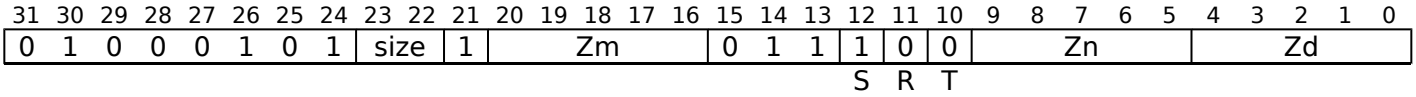
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SUBHNB

Subtract narrow high part (bottom)

Subtract each vector element of the second source vector from the corresponding vector element in the first source vector, and place the most significant half of the result in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. This instruction is unpredicated.



SUBHNB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	B
10	H
11	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	H
10	S
11	D

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;
constant integer halfesize = esize DIV 2;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer res = (element1 - element2) >> halfesize;
    Elem[result, 2*e + 0, halfesize] = res<halfesize-1:0>;
    Elem[result, 2*e + 1, halfesize] = Zeros(halfesize);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

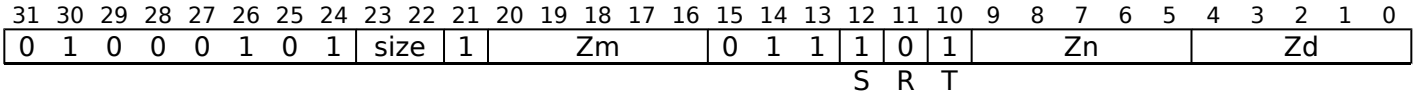
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SUBHNT

Subtract narrow high part (top)

Subtract each vector element of the second source vector from the corresponding vector element in the first source vector, and place the most significant half of the result in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. This instruction is unpredicated.



SUBHNT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	B
10	H
11	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	H
10	S
11	D

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[d, VL];
constant integer halfesize = esize DIV 2;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer res = (element1 - element2) >> halfesize;
    Elem[result, 2*e + 1, halfesize] = res<halfesize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

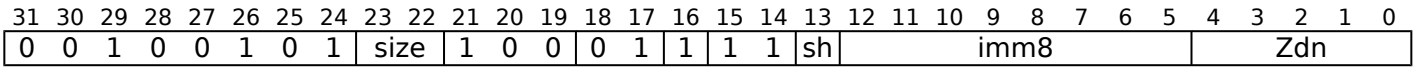
SUBR (immediate)

Reversed subtract from immediate (unpredicated)

Reversed subtract from an unsigned immediate each element of the source vector, and destructively place the results in the corresponding elements of the source vector. This instruction is unpredicated.

The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.

The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<uimm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".



SUBR <Zdn>.<T>, <Zdn>.<T>, #<imm>{, <shift>}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = UInt(imm8);
if sh == '1' then imm = imm << 8;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<imm> Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #8

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    Elem[result, e, esize] = (imm - element1)<esize-1:0>;

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.

    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

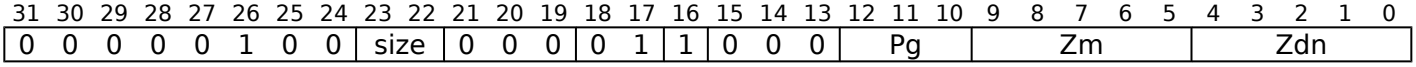
- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SUBR (vectors)

Reversed subtract vectors (predicated)

Reversed subtract active elements of the first source vector from corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



SUBR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = element2 - element1;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

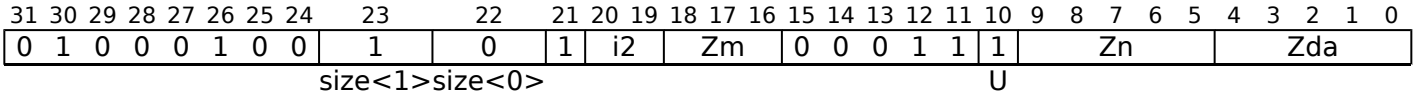
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SUDOT

Signed by unsigned integer indexed dot product

The signed by unsigned integer indexed dot product instruction computes the dot product of a group of four signed 8-bit integer values held in each 32-bit element of the first source vector multiplied by a group of four unsigned 8-bit integer values in an indexed 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.
The groups within the second source vector are specified using an immediate index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3. This instruction is unpredicated.
ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

SVE
(FEAT_I8MM)



SUDOT <Zda>.S, <Zn>.B, <Zm>.B[<imm>]

```
if (!HaveSVE() && !HaveSME()) || !HaveInt8MatMulExt() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
<imm>	Is the immediate index of a 32-bit group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 3
        integer element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        integer element2 = UInt(Elem[operand2, 4 * s + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = res;

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

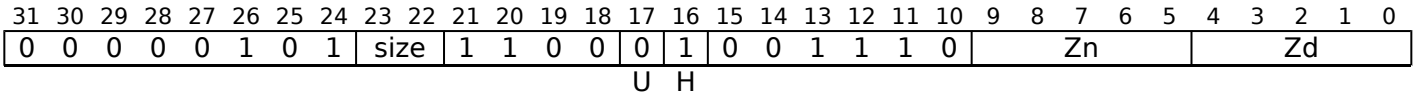
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SUNPKHI, SUNPKLO

Signed unpack and extend half of vector

Unpack elements from the lowest or highest half of the source vector and then sign-extend them to place in elements of twice their size within the destination vector. This instruction is unpredicated.
It has encodings from 2 classes: High half and Low half

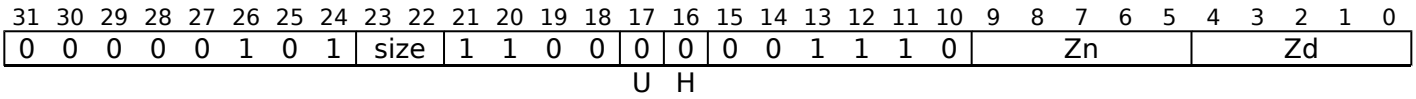
High half



SUNPKHI <Zd>.<T>, <Zn>.<Tb>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = FALSE;
boolean hi = TRUE;
```

Low half



SUNPKLO <Zd>.<T>, <Zn>.<Tb>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = FALSE;
boolean hi = FALSE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer hsize = esize DIV 2;
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(hsize) element = if hi then Elem[operand, e + elements, hsize] else Elem[operand, e, hsize];
    Elem[result, e, esize] = Extend(element, esize, unsigned);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

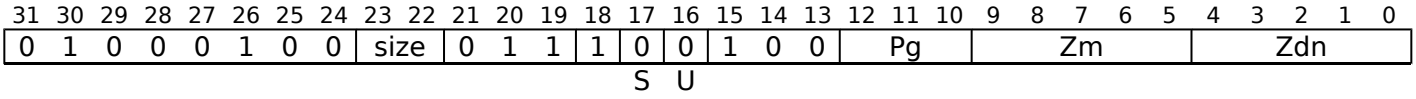
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SUQADD

Signed saturating addition of unsigned value

Add active unsigned elements of the source vector to the corresponding signed elements of the addend vector, and destructively place the results in the corresponding elements of the addend vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. Inactive elements in the destination vector register remain unmodified.



SUQADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = SignedSat(SInt(element1) + UInt(element2), esize);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

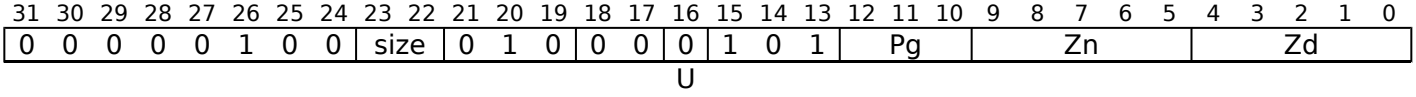
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

SXTB, SXTH, SXTW

Signed byte / halfword / word extend (predicated)

Sign-extend the least-significant sub-element of each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.
It has encodings from 3 classes: Byte , Halfword and Word

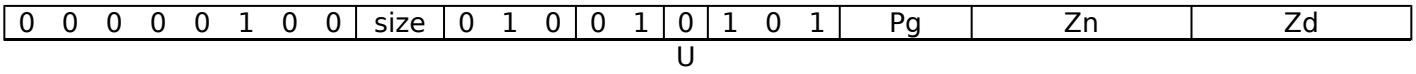
Byte



SXTB <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer s_esize = 8;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = FALSE;
```

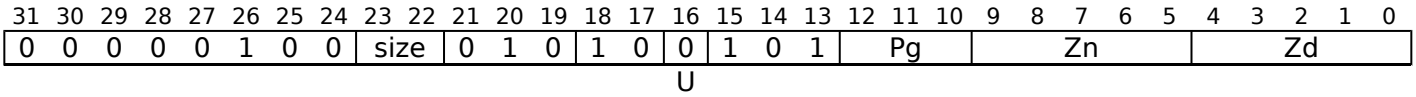
Halfword



SXTH <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer s_esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = FALSE;
```

Word



SXTW <Zd>.D, <Pg>/M, <Zn>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size != '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer s_esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = FALSE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> For the byte variant: is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

For the halfword variant: is the size specifier, encoded in “size<0>”:

size<0>	<T>
0	S
1	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = Extend(element<s_esize-1:0>, esize, unsigned);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

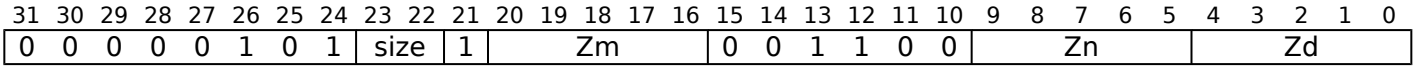
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

TBL

Programmable table lookup in one or two vector table (zeroing)

Reads each element of the second source (index) vector and uses its value to select an indexed element from a table of elements consisting of one or two consecutive vector registers, where the first or only vector holds the lower numbered elements, and places the indexed table element in the destination vector element corresponding to the index vector element. If an index value is greater than or equal to the number of vector elements then it places zero in the corresponding destination vector element.
Since the index values can select any element in a vector this operation is not naturally vector length agnostic.
It has encodings from 2 classes: SVE and SVE2

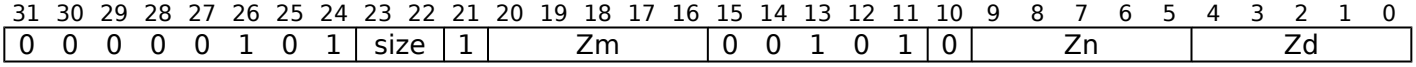
SVE



TBL <Zd>.<T>, { <Zn>.<T> }, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean double_table = FALSE;
```

SVE2



TBL <Zd>.<T>, { <Zn1>.<T>, <Zn2>.<T> }, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean double_table = TRUE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zn1> Is the name of the first scalable vector register of a multi-vector sequence, encoded in the "Zn" field.

<Zn2> Is the name of the second scalable vector register of a multi-vector sequence, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) indexes = Z[m, VL];
bits(VL) result;
constant integer table_size = if double_table then VL*2 else VL;
constant integer table_elems = table_size DIV esize;
bits(table_size) table;

if double_table then
    bits(VL) top = Z[(n + 1) MOD 32, VL];
    bits(VL) bottom = Z[n, VL];
    table = (top:bottom)<table_size-1:0>;
else
    table = Z[n, table_size];

for e = 0 to elements-1
    integer idx = UInt(Elem[indexes, e, esize]);
    Elem[result, e, esize] = if idx < table_elems then Elem[table, idx, esize] else Zeros(esize);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

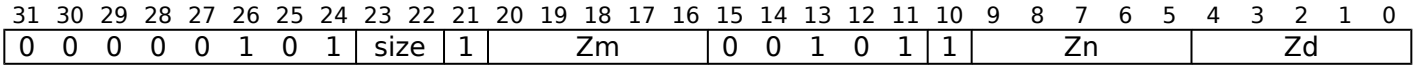
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

TBX

Programmable table lookup in single vector table (merging)

Reads each element of the second source (index) vector and uses its value to select an indexed element from a table of elements in the first source vector, and places the indexed element in the destination vector element corresponding to the index vector element. If an index value is greater than or equal to the number of vector elements then the corresponding destination vector element is left unchanged.
Since the index values can select any element in a vector this operation is not naturally vector length agnostic.



TBX <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    integer element2 = UInt(Elem[operand2, e, esize]);
    if element2 < elements then
        Elem[result, e, esize] = Elem[operand1, element2, esize];

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

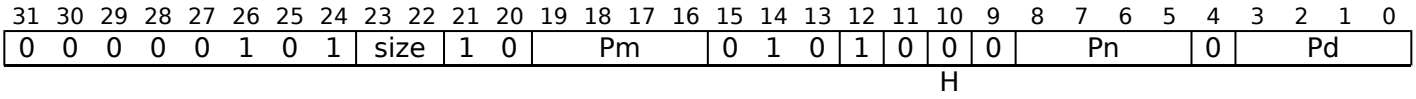
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

TRN1, TRN2 (predicates)

Interleave even or odd elements from two predicates

Interleave alternating even or odd-numbered elements from the first and second source predicates and place in elements of the destination predicate. This instruction is unpredicated.
It has encodings from 2 classes: Even and Odd

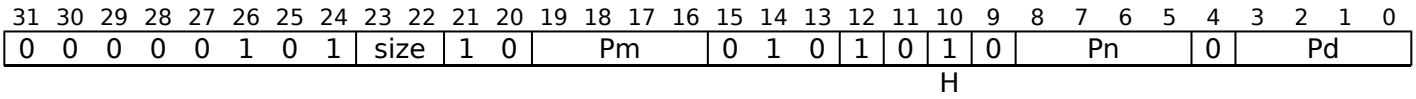
Even



TRN1 <Pd>.<T>, <Pn>.<T>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
integer part = 0;
```

Odd



TRN2 <Pd>.<T>, <Pn>.<T>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
integer part = 1;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (esize * 2);
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize DIV 8] = Elem[operand1, 2*p+part, esize DIV 8];
    Elem[result, 2*p+1, esize DIV 8] = Elem[operand2, 2*p+part, esize DIV 8];

P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

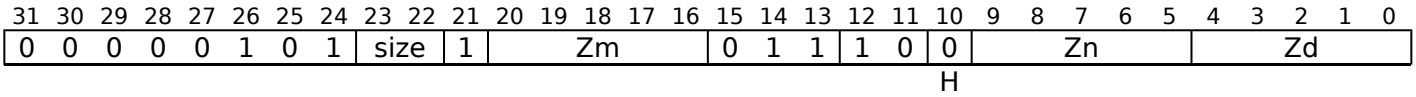
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

TRN1, TRN2 (vectors)

Interleave even or odd elements from two vectors

Interleave alternating even or odd-numbered elements from the first and second source vectors and place in elements of the destination vector. This instruction is unpredicated. The 128-bit element variant of this instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits are set to zero.
ID_AA64ZFR0_EL1.F64MM indicates whether the 128-bit element variant of the instruction is implemented.
It has encodings from 4 classes: Even , Even (quadwords) , Odd and Odd (quadwords)

Even

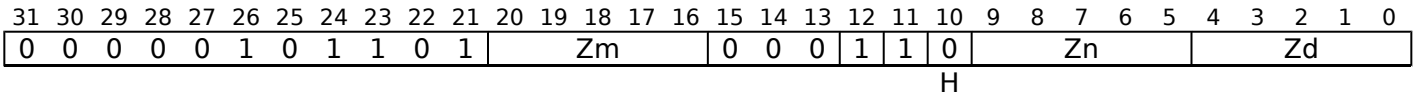


TRN1 <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;
```

Even (quadwords)

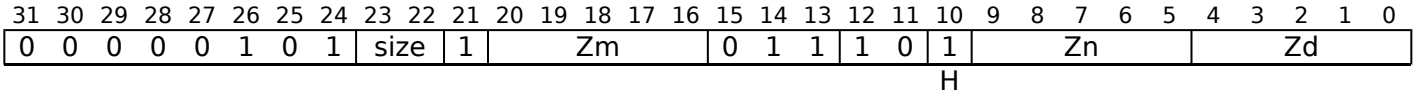
(FEAT_F64MM)



TRN1 <Zd>.Q, <Zn>.Q, <Zm>.Q

```
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
constant integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;
```

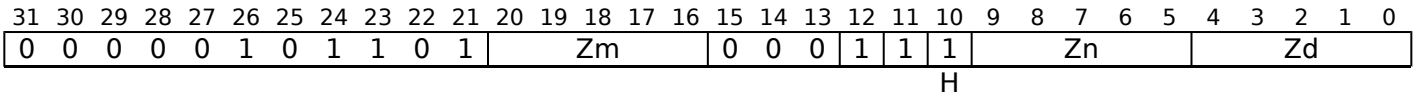
Odd



TRN2 <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;
```

Odd (quadwords)
(FEAT_F64MM)



TRN2 <Zd>.Q, <Zn>.Q, <Zm>.Q

```
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
constant integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
if esize < 128 then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
if VL < esize * 2 then UNDEFINED;
constant integer pairs = VL DIV (esize * 2);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Zeros(VL);

for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, 2*p+part, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, 2*p+part, esize];

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

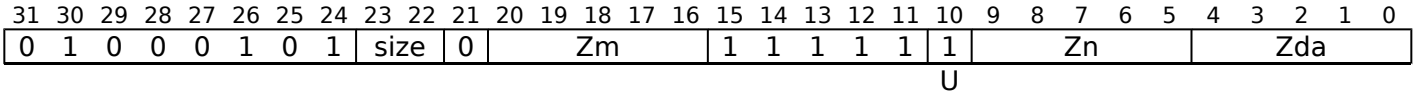
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UABA

Unsigned absolute difference and accumulate

Compute the absolute difference between unsigned integer values in elements of the second source vector and corresponding elements of the first source vector, and add the difference to the corresponding elements of the destination vector. This instruction is unpredicated.



UABA <Zda>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean unsigned = TRUE;
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    bits(esize) absdiff = Abs(element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

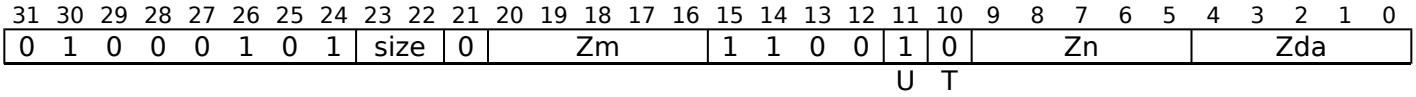
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UABALB

Unsigned absolute difference and accumulate long (bottom)

Compute the absolute difference between even-numbered unsigned elements of the second source vector and corresponding elements of the first source vector, and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.



UABALB <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, 2*e + 0, esize DIV 2]);
    integer element2 = UInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    bits(esize) absdiff = Abs(element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

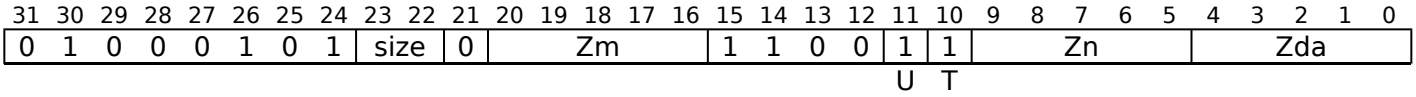
- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UABALT

Unsigned absolute difference and accumulate long (top)

Compute the absolute difference between odd-numbered unsigned elements of the second source vector and corresponding elements of the first source vector, and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.



UABALT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, 2*e + 1, esize DIV 2]);
    integer element2 = UInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    bits(esize) absdiff = Abs(element1 - element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + absdiff;

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

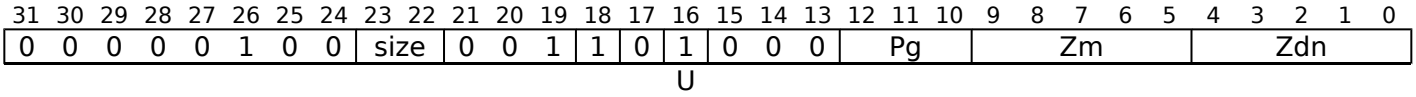
- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UABD

Unsigned absolute difference (predicated)

Compute the absolute difference between unsigned integer values in active elements of the second source vector and corresponding elements of the first source vector and destructively place the difference in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



UABD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    if ElemP[mask, e, esize] == '1' then
        integer absdiff = Abs(element1 - element2);
        Elem[result, e, esize] = absdiff<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

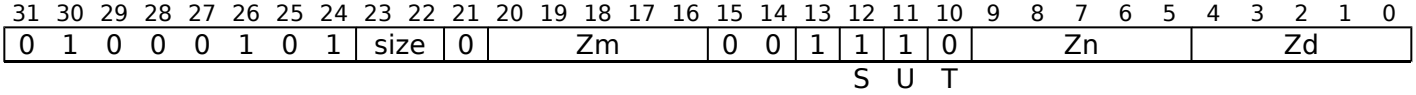
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UABDLB

Unsigned absolute difference long (bottom)

Compute the absolute difference between the even-numbered unsigned integer values in elements of the second source vector and the corresponding elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



UABDLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, 2*e + 0, esize DIV 2]);
    integer element2 = UInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    integer res = Abs(element1 - element2);
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

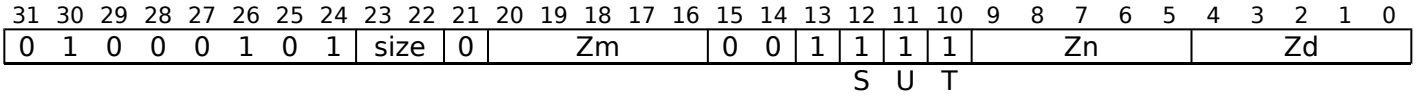
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UABDLT

Unsigned absolute difference long (top)

Compute the absolute difference between the odd-numbered unsigned integer values in elements of the second source vector and corresponding elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



UABDLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, 2*e + 1, esize DIV 2]);
    integer element2 = UInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    integer res = Abs(element1 - element2);
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

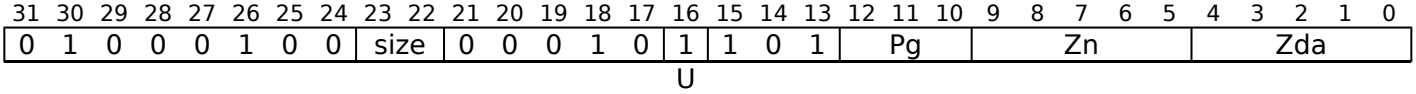
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UADALP

Unsigned add and accumulate long pairwise

Add pairs of adjacent unsigned integer values and accumulate the results into the overlapping double-width elements of the destination vector.



UADALP <Zda>.<T>, <Pg>/M, <Zn>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the second source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand_acc = Z[da, VL];
bits(VL) operand_src = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '0' then
        Elem[result, e, esize] = Elem[operand_acc, e, esize];
    else
        integer element1 = UInt(Elem[operand_src, 2*e + 0, esize DIV 2]);
        integer element2 = UInt(Elem[operand_src, 2*e + 1, esize DIV 2]);
        bits(esize) sum = (element1 + element2)<esize-1:0>;
        Elem[result, e, esize] = Elem[operand_acc, e, esize] + sum;

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

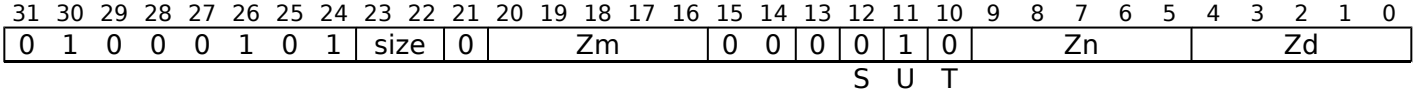
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UADDLB

Unsigned add long (bottom)

Add the corresponding even-numbered unsigned elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



UADDLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 0;
integer sel2 = 0;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 + element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

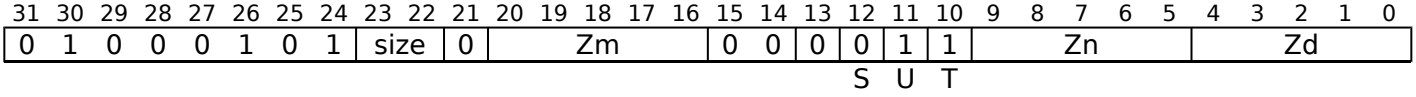
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UADDLT

Unsigned add long (top)

Add the corresponding odd-numbered unsigned elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



UADDLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 1;
integer sel2 = 1;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 + element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

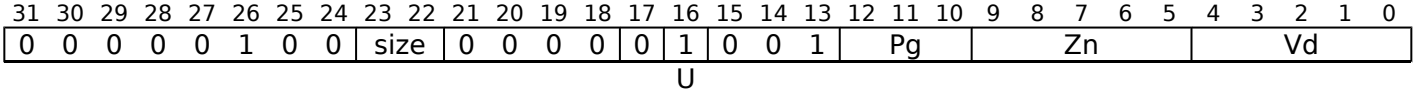
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UADDV

Unsigned add reduction to scalar

Unsigned add horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Narrow elements are first zero-extended to 64 bits. Inactive elements in the source vector are treated as zero.



UADDV <Dd>, <Pg>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
```

Assembler Symbols

<Dd> Is the 64-bit name of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
integer sum = 0;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element = UInt(Elem[operand, e, esize]);
        sum = sum + element;

V[d, 64] = sum<63:0>;
```

Operational information

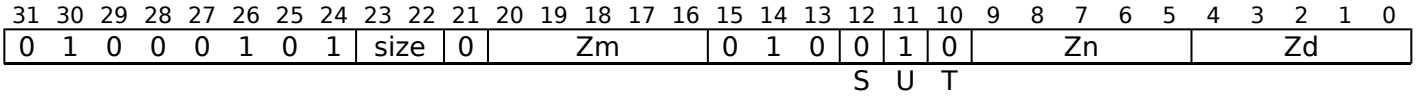
If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

UADDWB

Unsigned add wide (bottom)

Add the even-numbered unsigned elements of the second source vector to the overlapping double-width elements of the first source vector and place the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.



UADDWB <Zd>.<T>, <Zn>.<T>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    Elem[result, e, esize] = (element1 + element2)<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

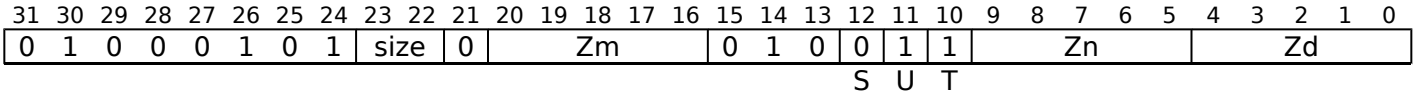
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UADDWT

Unsigned add wide (top)

Add the odd-numbered unsigned elements of the second source vector to the overlapping double-width elements of the first source vector and place the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.



UADDWT <Zd>.<T>, <Zn>.<T>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    Elem[result, e, esize] = (element1 + element2)<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

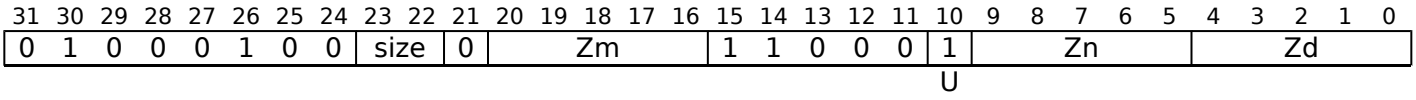
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UCLAMP

Unsigned clamp to minimum/maximum vector

Clamp each unsigned element in the destination vector to between the unsigned minimum value in the corresponding element of the first source vector and the unsigned maximum value in the corresponding element of the second source vector and destructively write the results in the corresponding elements of the destination vector. This instruction is unpredicated.

SVE2
(FEAT_SME)



UCLAMP <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[d, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    integer element3 = UInt(Elem[operand3, e, esize]);
    integer res = Min(Max(element1 , element3), element2);
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

        - The values of the data supplied in any of its registers.
        - The values of the NZCV flags.
    - The response of this instruction to asynchronous exceptions does not vary based on:
        - The values of the data supplied in any of its registers.
        - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

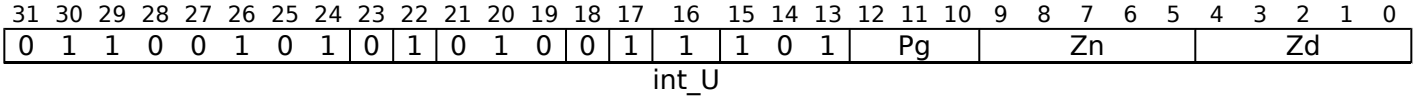
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UCVTF

Unsigned integer convert to floating-point (predicated)

Convert to floating-point from the unsigned integer in each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.
If the input and result types have a different size the smaller type is held unpacked in the least significant bits of elements of the larger size. When the input is the smaller type the upper bits of each source element are ignored. When the result is the smaller type the results are zero-extended to fill each destination element.
It has encodings from 7 classes: 16-bit to half-precision , 32-bit to half-precision , 32-bit to single-precision , 32-bit to double-precision , 64-bit to half-precision , 64-bit to single-precision and 64-bit to double-precision

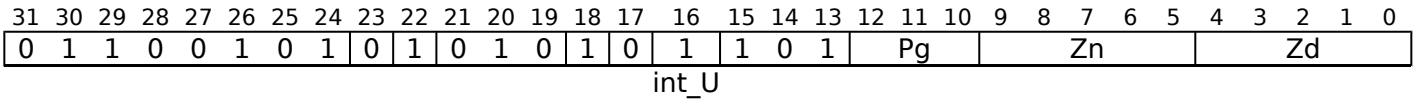
16-bit to half-precision



UCVTF <Zd>.H, <Pg>/M, <Zn>.H

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 16;
constant integer d_esize = 16;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

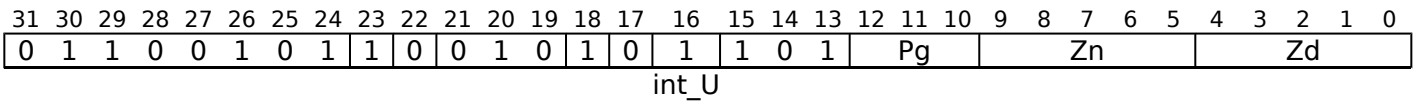
32-bit to half-precision



UCVTF <Zd>.H, <Pg>/M, <Zn>.S

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 32;
constant integer d_esize = 16;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

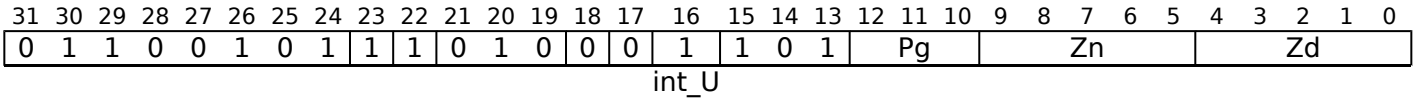
32-bit to single-precision



UCVTF <Zd>.S, <Pg>/M, <Zn>.S

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 32;
constant integer d_esize = 32;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

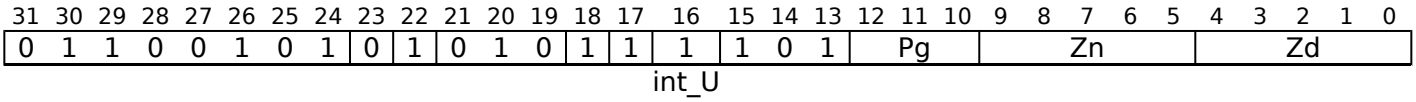
32-bit to double-precision



UCVTF <Zd>.D, <Pg>/M, <Zn>.S

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 32;
constant integer d_esize = 64;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

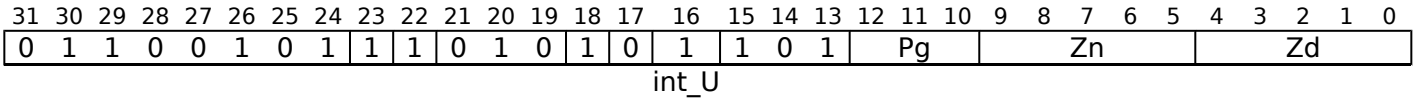
64-bit to half-precision



UCVTF <Zd>.H, <Pg>/M, <Zn>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 64;
constant integer d_esize = 16;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

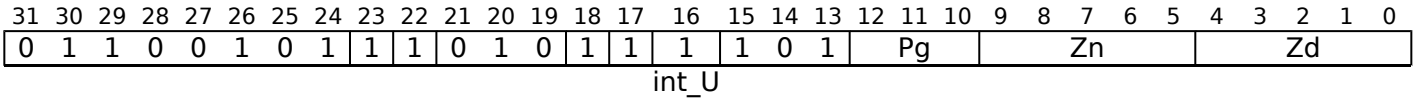
64-bit to single-precision



UCVTF <Zd>.S, <Pg>/M, <Zn>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 64;
constant integer d_esize = 32;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

64-bit to double-precision



UCVTF <Zd>.D, <Pg>/M, <Zn>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
constant integer s_esize = 64;
constant integer d_esize = 64;
boolean unsigned = TRUE;
FPRounding rounding = FPRoundingMode(FPCR[]);
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        bits(d_esize) fpval = FixedToFP(element<s_esize-1:0>, 0, unsigned, FPCR[], rounding, d_esize);
        Elem[result, e, esize] = ZeroExtend(fpval, esize);

Z[d, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.

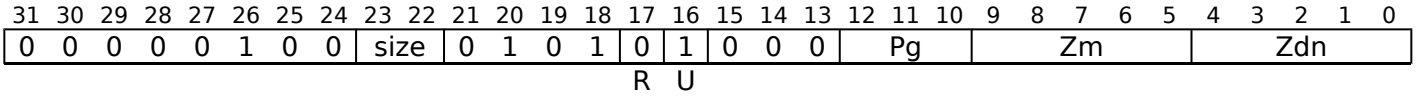
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UDIV

Unsigned divide (predicated)

Unsigned divide active elements of the first source vector by corresponding elements of the second source vector and destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



UDIV <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size<0>”:

size<0>	<T>
0	S
1	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    if ElemP[mask, e, esize] == '1' then
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        integer quotient;
        if element2 == 0 then
            quotient = 0;
        else
            quotient = RoundTowardsZero(Real(element1) / Real(element2));
        Elem[result, e, esize] = quotient<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

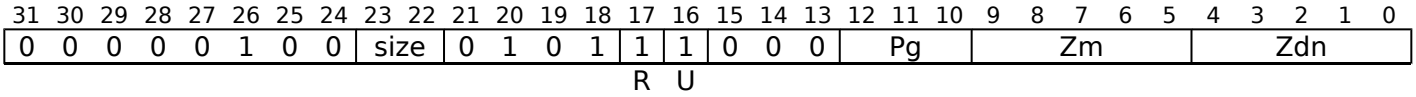
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UDIVR

Unsigned reversed divide (predicated)

Unsigned reversed divide active elements of the second source vector by corresponding elements of the first source vector and destructively place the quotient in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



UDIVR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size<0>”:

size<0>	<T>
0	S
1	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    if ElemP[mask, e, esize] == '1' then
        integer element2 = Int(Elem[operand2, e, esize], unsigned);
        integer quotient;
        if element1 == 0 then
            quotient = 0;
        else
            quotient = RoundTowardsZero(Real(element2) / Real(element1));
        Elem[result, e, esize] = quotient<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

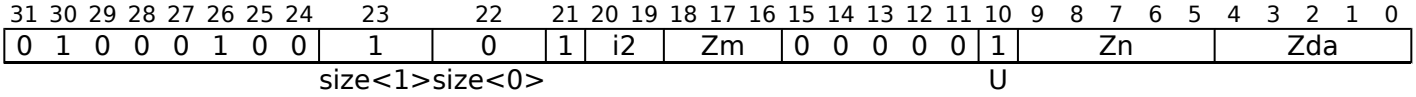
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UDOT (indexed)

Unsigned integer indexed dot product

The unsigned integer indexed dot product instruction computes the dot product of a group of four unsigned 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the first source vector multiplied by a group of four unsigned 8-bit or 16-bit integer values in an indexed 32-bit or 64-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit or 64-bit element of the destination vector. The groups within the second source vector are specified using an immediate index which selects the same group position within each 128-bit vector segment. The index range is from 0 to one less than the number of groups per 128-bit segment, encoded in 1 to 2 bits depending on the size of the group. This instruction is unpredicated.
It has encodings from 2 classes: 32-bit and 64-bit

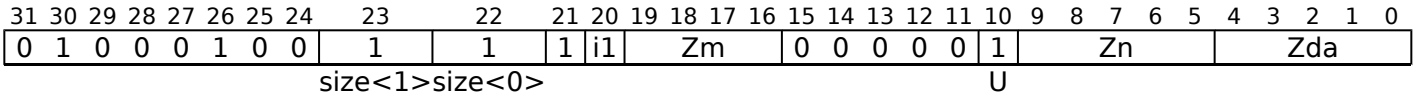
32-bit



UDOT <Zda>.S, <Zn>.B, <Zm>.B[<imm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

64-bit



UDOT <Zda>.D, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field. For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the 32-bit variant: is the immediate index of a 32-bit group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field. For the 64-bit variant: is the immediate index of a 64-bit group of four 16-bit values within each 128-bit vector segment, in the range 0 to 1, encoded in the "i1" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 3
        integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        integer element2 = UInt(Elem[operand2, 4 * s + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = res;

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

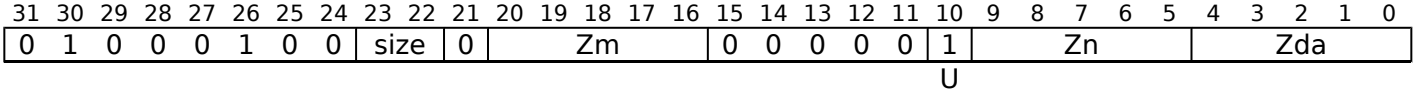
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UDOT (vectors)

Unsigned integer dot product

The unsigned integer dot product instruction computes the dot product of a group of four unsigned 8-bit or 16-bit integer values held in each 32-bit or 64-bit element of the first source vector multiplied by a group of four unsigned 8-bit or 16-bit integer values in the corresponding 32-bit or 64-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit or 64-bit element of the destination vector. This instruction is unpredicated.



UDOT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size<0>”:

size<0>	<T>
0	S
1	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size<0>”:

size<0>	<Tb>
0	B
1	H

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 3
        integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        integer element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = res;

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

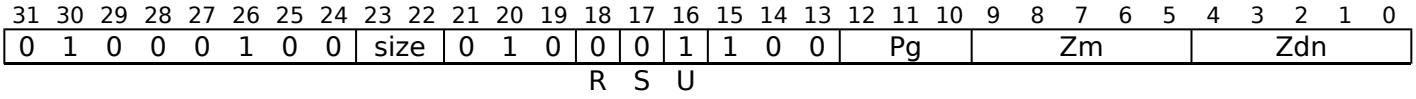
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UHADD

Unsigned halving addition

Add active unsigned elements of the first source vector to corresponding unsigned elements of the second source vector, shift right one bit, and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



UHADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    if ElemP[mask, e, esize] == '1' then
        integer res = (element1 + element2) >> 1;
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

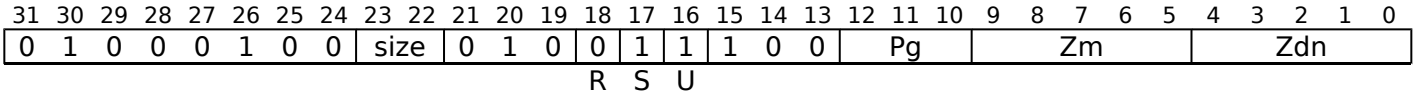
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UHSUB

Unsigned halving subtract

Subtract active unsigned elements of the second source vector from corresponding unsigned elements of the first source vector, shift right one bit, and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



UHSUB <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    if ElemP[mask, e, esize] == '1' then
        integer res = (element1 - element2) >> 1;
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

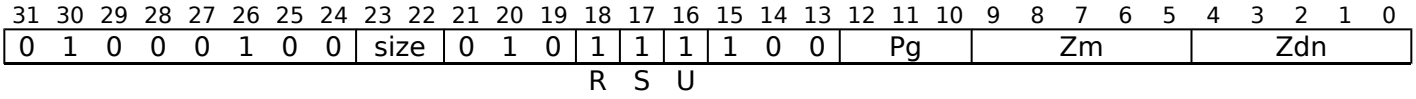
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UHSUBR

Unsigned halving subtract reversed vectors

Subtract active unsigned elements of the first source vector from corresponding unsigned elements of the second source vector, shift right one bit, and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



UHSUBR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    if ElemP[mask, e, esize] == '1' then
        integer res = (element2 - element1) >> 1;
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.

    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

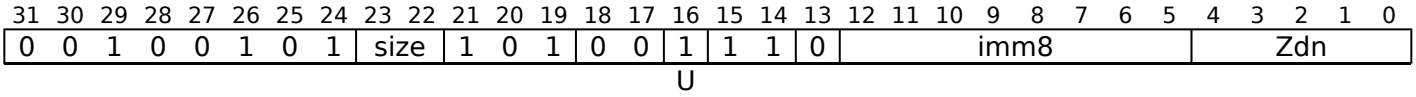
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMAX (immediate)

Unsigned maximum with immediate (unpredicated)

Determine the unsigned maximum of an immediate and each element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is an unsigned 8-bit value in the range 0 to 255, inclusive. This instruction is unpredicated.



UMAX <Zdn>.<T>, <Zdn>.<T>, #<imm>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
boolean unsigned = TRUE;
integer imm = Int(imm8, unsigned);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<imm> Is the unsigned immediate operand, in the range 0 to 255, encoded in the "imm8" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    Elem[result, e, esize] = Max(element1, imm)<esize-1:0>;

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

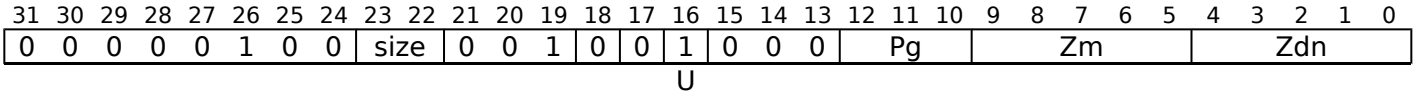
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

UMAX (vectors)

Unsigned maximum vectors (predicated)

Determine the unsigned maximum of active elements of the second source vector and corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



UMAX <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    if ElemP[mask, e, esize] == '1' then
        integer maximum = Max(element1, element2);
        Elem[result, e, esize] = maximum<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

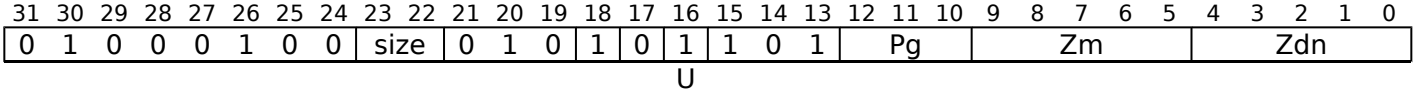
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMAXP

Unsigned maximum pairwise

Compute the maximum value of each pair of adjacent unsigned integer elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.



UMAXP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
integer element1;
integer element2;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '0' then
        Elem[result, e, esize] = Elem[operand1, e, esize];
    else
        if IsEven(e) then
            element1 = UInt(Elem[operand1, e + 0, esize]);
            element2 = UInt(Elem[operand1, e + 1, esize]);
        else
            element1 = UInt(Elem[operand2, e - 1, esize]);
            element2 = UInt(Elem[operand2, e + 0, esize]);
        integer res = Max(element1, element2);
        Elem[result, e, esize] = res<esize-1:0>;

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

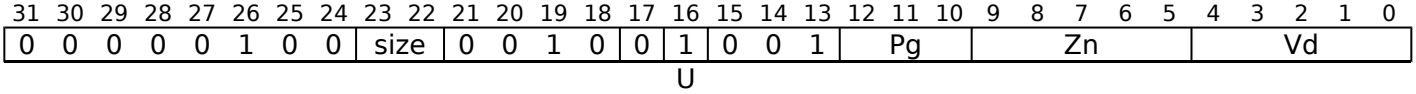
- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMAXV

Unsigned maximum reduction to scalar

Unsigned maximum horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as zero.



UMAXV <V><d>, <Pg>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean unsigned = TRUE;
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
integer maximum = if unsigned then 0 else -(2^(esize-1));

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element = Int(Elem[operand, e, esize], unsigned);
        maximum = Max(maximum, element);

V[d, esize] = maximum<esize-1:0>;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

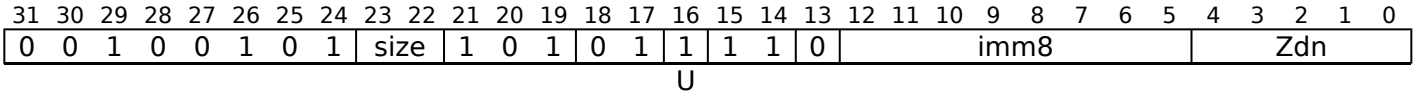
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMIN (immediate)

Unsigned minimum with immediate (unpredicated)

Determine the unsigned minimum of an immediate and each element of the source vector, and destructively place the results in the corresponding elements of the source vector. The immediate is an unsigned 8-bit value in the range 0 to 255, inclusive. This instruction is unpredicated.




```
UMIN <Zdn>.<T>, <Zdn>.<T>, #<imm>
```

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
boolean unsigned = TRUE;
integer imm = Int(imm8, unsigned);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<imm> Is the unsigned immediate operand, in the range 0 to 255, encoded in the "imm8" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    Elem[result, e, esize] = Min(element1, imm)<esize-1:0>;

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

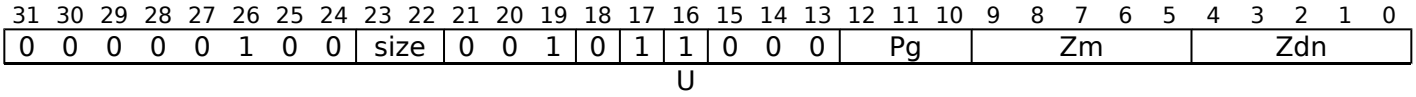
This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

UMIN (vectors)

Unsigned minimum vectors (predicated)

Determine the unsigned minimum of active elements of the second source vector and corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



UMIN <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    if ElemP[mask, e, esize] == '1' then
        integer minimum = Min(element1, element2);
        Elem[result, e, esize] = minimum<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

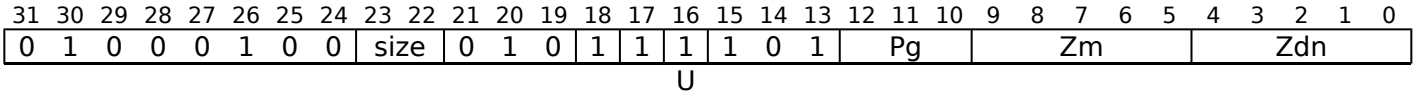
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMINP

Unsigned minimum pairwise

Compute the minimum value of each pair of adjacent unsigned integer elements within each source vector, and interleave the results from corresponding lanes. The interleaved result values are destructively placed in the first source vector.



UMINP <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
integer element1;
integer element2;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '0' then
        Elem[result, e, esize] = Elem[operand1, e, esize];
    else
        if IsEven(e) then
            element1 = UInt(Elem[operand1, e + 0, esize]);
            element2 = UInt(Elem[operand1, e + 1, esize]);
        else
            element1 = UInt(Elem[operand2, e - 1, esize]);
            element2 = UInt(Elem[operand2, e + 0, esize]);
        integer res = Min(element1, element2);
        Elem[result, e, esize] = res<esize-1:0>;

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

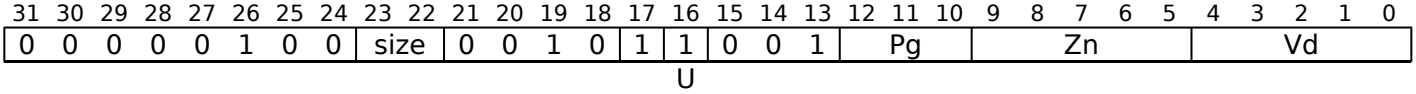
- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMINV

Unsigned minimum reduction to scalar

Unsigned minimum horizontally across all lanes of a vector, and place the result in the SIMD&FP scalar destination register. Inactive elements in the source vector are treated as the maximum unsigned integer for the element size.



UMINV <V><d>, <Pg>, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Vd);
boolean unsigned = TRUE;
```

Assembler Symbols

<V> Is a width specifier, encoded in “size”:

size	<V>
00	B
01	H
10	S
11	D

<d> Is the number [0-31] of the destination SIMD&FP register, encoded in the "Vd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
integer minimum = if unsigned then (2^esize - 1) else (2^(esize-1) - 1);

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element = Int(Elem[operand, e, esize], unsigned);
        minimum = Min(minimum, element);

V[d, esize] = minimum<esize-1:0>;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

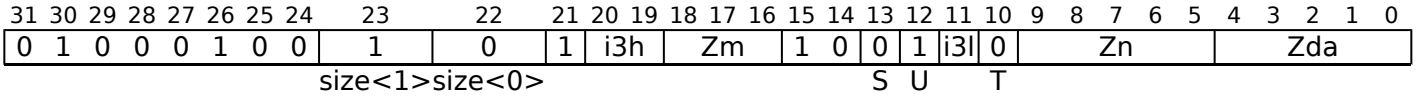
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMLALB (indexed)

Unsigned multiply-add long to accumulator (bottom, indexed)

Multiply the even-numbered unsigned elements within each 128-bit segment of the first source vector by the specified unsigned element in the corresponding second source vector segment and destructively add to the overlapping double-width elements of the addend vector.
The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.
It has encodings from 2 classes: 32-bit and 64-bit

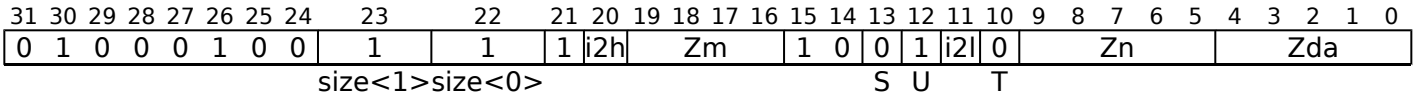
32-bit



UMLALB <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```

64-bit



UMLALB <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field. For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields. For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = UInt(Elem[operand1, 2 * e + sel,   esize]);
    integer element2 = UInt(Elem[operand2, 2 * s + index, esize]);
    bits(2*esize) product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + product;

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

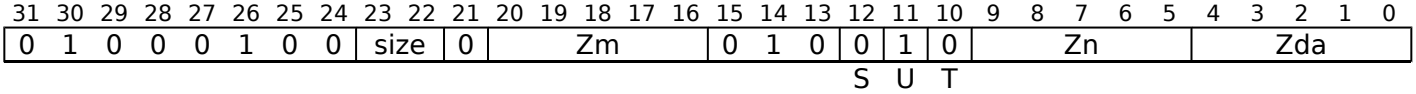
- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMLALB (vectors)

Unsigned multiply-add long to accumulator (bottom)

Multiply the corresponding even-numbered unsigned elements of the first and second source vectors and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.



UMLALB <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, 2*e + 0, esize DIV 2]);
    integer element2 = UInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    bits(esize) product = (element1 * element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + product;

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

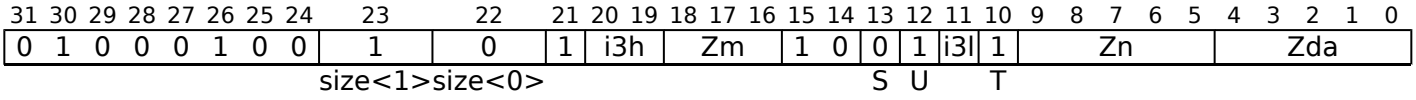
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMLALT (indexed)

Unsigned multiply-add long to accumulator (top, indexed)

Multiply the odd-numbered unsigned elements within each 128-bit segment of the first source vector by the specified unsigned element in the corresponding second source vector segment and destructively add to the overlapping double-width elements of the addend vector.
The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.
It has encodings from 2 classes: 32-bit and 64-bit

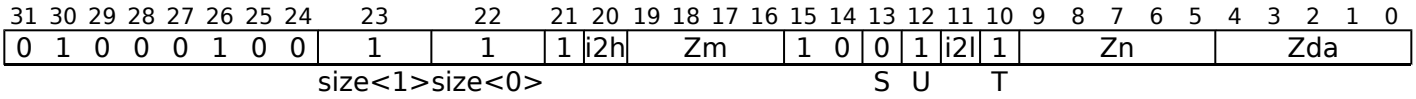
32-bit



UMLALT <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
```

64-bit



UMLALT <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field. For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields. For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = UInt(Elem[operand1, 2 * e + sel,   esize]);
    integer element2 = UInt(Elem[operand2, 2 * s + index, esize]);
    bits(2*esize) product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] + product;

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

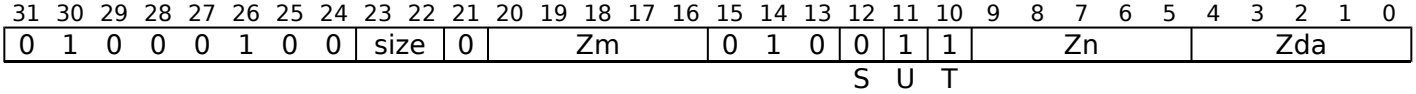
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMLALT (vectors)

Unsigned multiply-add long to accumulator (top)

Multiply the corresponding odd-numbered unsigned elements of the first and second source vectors and destructively add to the overlapping double-width elements of the addend vector. This instruction is unpredicated.



UMLALT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, 2*e + 1, esize DIV 2]);
    integer element2 = UInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    bits(esize) product = (element1 * element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] + product;

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

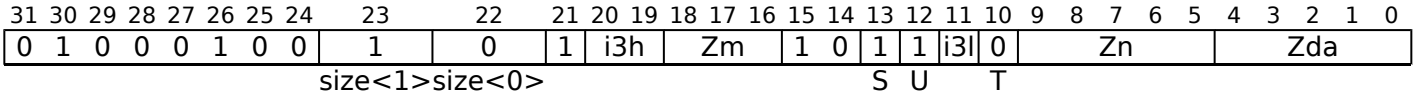
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMLSLB (indexed)

Unsigned multiply-subtract long from accumulator (bottom, indexed)

Multiply the even-numbered unsigned elements within each 128-bit segment of the first source vector by the specified unsigned element in the corresponding second source vector segment and destructively subtract from the overlapping double-width elements of the addend vector.
The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.
It has encodings from 2 classes: 32-bit and 64-bit

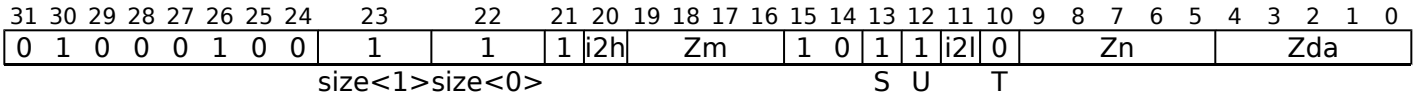
32-bit



UMLSLB <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```

64-bit



UMLSLB <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 0;
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field. For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields. For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = UInt(Elem[operand1, 2 * e + sel,   esize]);
    integer element2 = UInt(Elem[operand2, 2 * s + index, esize]);
    bits(2*esize) product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] - product;

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

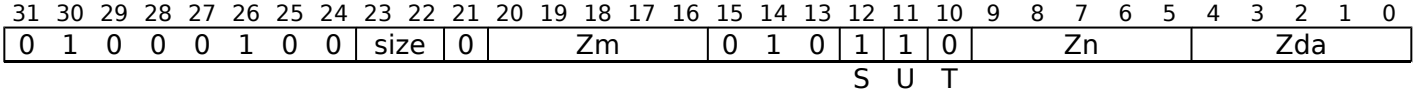
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMLSLB (vectors)

Unsigned multiply-subtract long from accumulator (bottom)

Multiply the corresponding even-numbered unsigned elements of the first and second source vectors and destructively subtract from the overlapping double-width elements of the addend vector. This instruction is unpredicated.



UMLSLB <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, 2*e + 0, esize DIV 2]);
    integer element2 = UInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    bits(esize) product = (element1 * element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] - product;

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

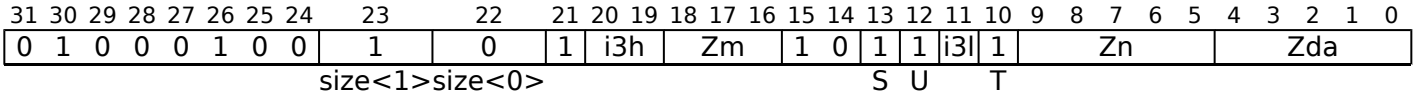
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMLSLT (indexed)

Unsigned multiply-subtract long from accumulator (top, indexed)

Multiply the odd-numbered unsigned elements within each 128-bit segment of the first source vector by the specified unsigned element in the corresponding second source vector segment and destructively subtract from the overlapping double-width elements of the addend vector.
The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.
It has encodings from 2 classes: 32-bit and 64-bit

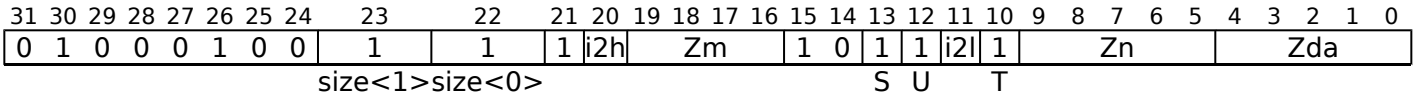
32-bit



UMLSLT <Zda>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
```

64-bit



UMLSLT <Zda>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel = 1;
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field. For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields. For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = UInt(Elem[operand1, 2 * e + sel,   esize]);
    integer element2 = UInt(Elem[operand2, 2 * s + index, esize]);
    bits(2*esize) product = (element1 * element2)<2*esize-1:0>;
    Elem[result, e, 2*esize] = Elem[result, e, 2*esize] - product;

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

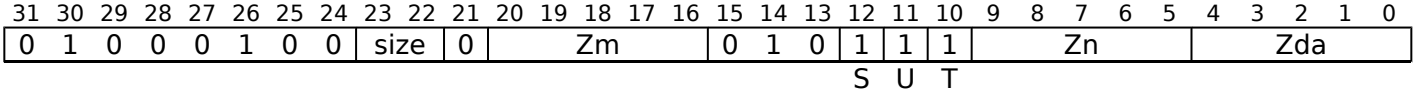
- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMLSLT (vectors)

Unsigned multiply-subtract long from accumulator (top)

Multiply the corresponding odd-numbered unsigned elements of the first and second source vectors and destructively subtract from the overlapping double-width elements of the addend vector. This instruction is unpredicated.



UMLSLT <Zda>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Z[da, VL];

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, 2*e + 1, esize DIV 2]);
    integer element2 = UInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    bits(esize) product = (element1 * element2)<esize-1:0>;
    Elem[result, e, esize] = Elem[result, e, esize] - product;

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMMLA

Unsigned integer matrix multiply-accumulate

The unsigned integer matrix multiply-accumulate instruction multiplies the 2×8 matrix of unsigned 8-bit integer values held in each 128-bit segment of the first source vector by the 8×2 matrix of unsigned 8-bit integer values in the corresponding segment of the second source vector. The resulting 2×2 widened 32-bit integer matrix product is then destructively added to the 32-bit integer matrix accumulator held in the corresponding segment of the addend and destination vector. This is equivalent to performing an 8-way dot product per destination element.

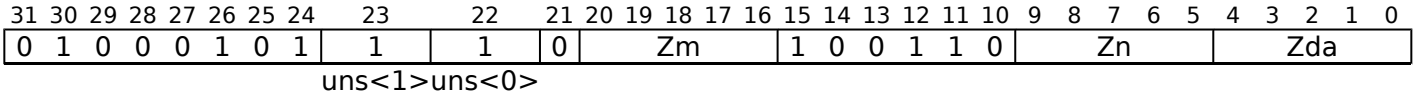
This instruction is unpredicated.

ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE

(FEAT_I8MM)



UMMLA <Zda>.S, <Zn>.B, <Zm>.B

```
if !HaveSVE() || !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_unsigned = TRUE;
boolean op2_unsigned = TRUE;
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer segments = VL DIV 128;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result = Zeros(VL);
bits(128) op1, op2;
bits(128) res, addend;

for s = 0 to segments-1
    op1    = Elem[operand1, s, 128];
    op2    = Elem[operand2, s, 128];
    addend = Elem[operand3, s, 128];
    res    = MatMulAdd(addend, op1, op2, op1_unsigned, op2_unsigned);
    Elem[result, s, 128] = res;

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

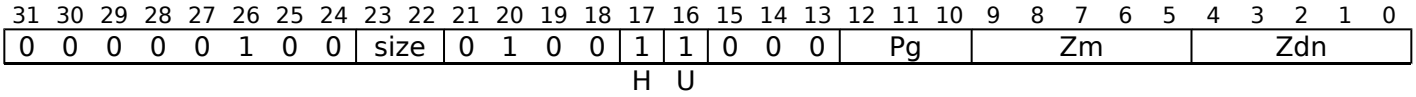
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMULH (predicated)

Unsigned multiply returning high half (predicated)

Widening multiply unsigned integer values in active elements of the first source vector by corresponding elements of the second source vector and destructively place the high half of the result in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



UMULH <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    if ElemP[mask, e, esize] == '1' then
        integer product = (element1 * element2) >> esize;
        Elem[result, e, esize] = product<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:

    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

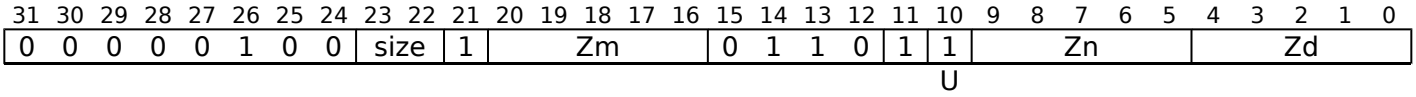
- The `MOVPRFX` instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMULH (unpredicated)

Unsigned multiply returning high half (unpredicated)

Widening multiply unsigned integer values of all elements of the first source vector by corresponding elements of the second source vector and place the high half of the result in the corresponding elements of the destination vector. This instruction is unpredicated.



UMULH <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean unsigned = TRUE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    integer product = (element1 * element2) >> esize;
    Elem[result, e, esize] = product<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

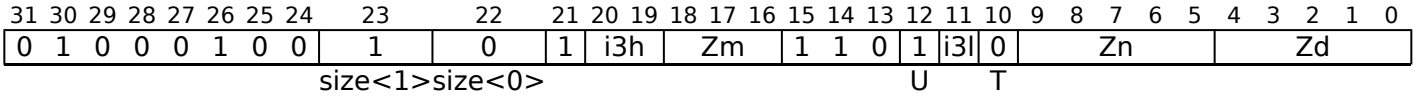
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

UMULLB (indexed)

Unsigned multiply long (bottom, indexed)

Multiply the even-numbered unsigned elements within each 128-bit segment of the first source vector by the specified unsigned element in the corresponding second source vector segment, and place the results in the overlapping double-width elements of the destination vector register.
The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.
It has encodings from 2 classes: 32-bit and 64-bit

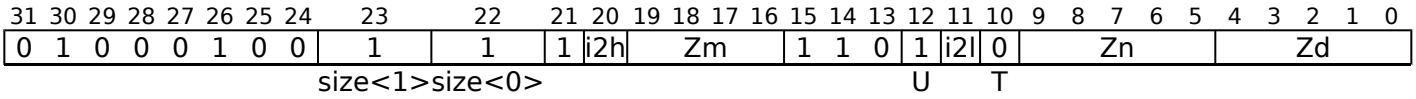
32-bit



UMULLB <Zd>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 0;
```

64-bit



UMULLB <Zd>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 0;
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field. For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields. For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = UInt(Elem[operand1, 2 * e + sel,   esize]);
    integer element2 = UInt(Elem[operand2, 2 * s + index, esize]);
    integer res = element1 * element2;
    Elem[result, e, 2*esize] = res<2*esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

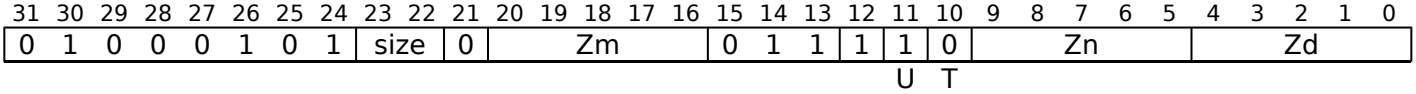
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMULLB (vectors)

Unsigned multiply long (bottom)

Multiply the corresponding even-numbered unsigned elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



UMULLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, 2*e + 0, esize DIV 2]);
    integer element2 = UInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    integer res = element1 * element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

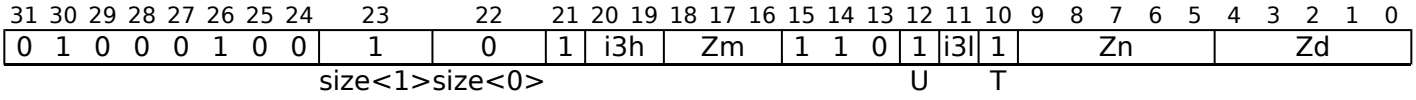
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMULLT (indexed)

Unsigned multiply long (top, indexed)

Multiply the odd-numbered unsigned elements within each 128-bit segment of the first source vector by the specified unsigned element in the corresponding second source vector segment, and place the results in the overlapping double-width elements of the destination vector register.
The elements within the second source vector are specified using an immediate index which selects the same element position within each 128-bit vector segment. The index range is from 0 to one less than the number of elements per 128-bit segment, encoded in 2 or 3 bits depending on the size of the element.
It has encodings from 2 classes: 32-bit and 64-bit

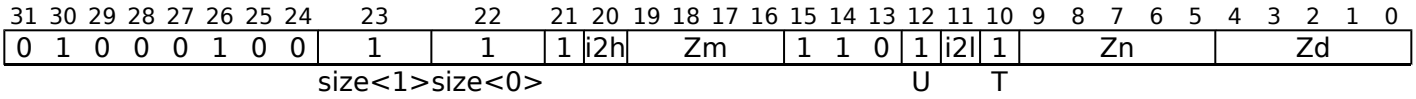
32-bit



UMULLT <Zd>.S, <Zn>.H, <Zm>.H[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i3h:i3l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 1;
```

64-bit



UMULLT <Zd>.D, <Zn>.S, <Zm>.S[<imm>]

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2h:i2l);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel = 1;
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	For the 32-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field. For the 64-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field.
<imm>	For the 32-bit variant: is the element index, in the range 0 to 7, encoded in the "i3h:i3l" fields. For the 64-bit variant: is the element index, in the range 0 to 3, encoded in the "i2h:i2l" fields.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
constant integer eltspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer s = e - (e MOD eltspersegment);
    integer element1 = UInt(Elem[operand1, 2 * e + sel,   esize]);
    integer element2 = UInt(Elem[operand2, 2 * s + index, esize]);
    integer res = element1 * element2;
    Elem[result, e, 2*esize] = res<2*esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

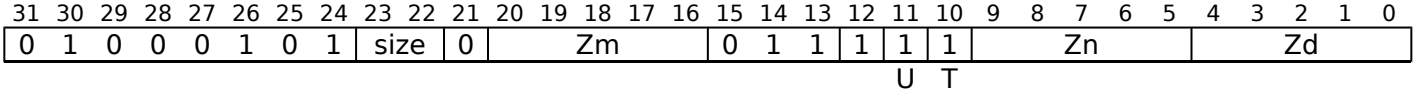
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UMULLT (vectors)

Unsigned multiply long (top)

Multiply the corresponding odd-numbered unsigned elements of the first and second source vectors, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



UMULLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, 2*e + 1, esize DIV 2]);
    integer element2 = UInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    integer res = element1 * element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

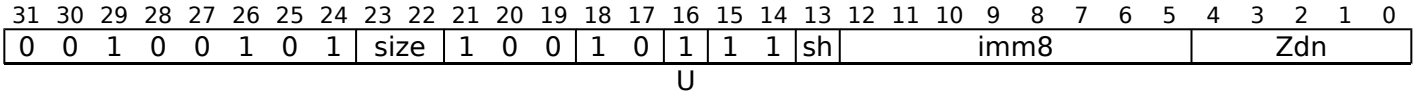
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQADD (immediate)

Unsigned saturating add immediate (unpredicated)

Unsigned saturating add of an unsigned immediate to each element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. This instruction is unpredicated.
The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.
The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<uimm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".



UQADD <Zdn>.<T>, <Zdn>.<T>, #<imm>{, <shift>}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = UInt(imm8);
if sh == '1' then imm = imm << 8;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<imm> Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #8

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + imm, esize, unsigned);

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

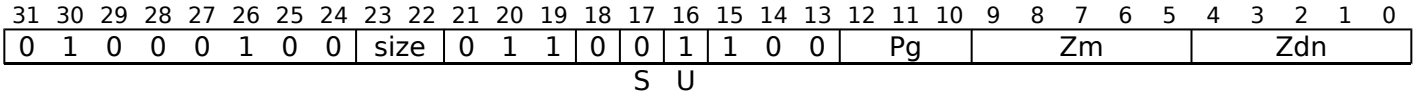
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQADD (vectors, predicated)

Unsigned saturating addition (predicated)

Add active unsigned elements of the first source vector to corresponding unsigned elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. Inactive elements in the destination vector register remain unmodified.



UQADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    if ElemP[mask, e, esize] == '1' then
        integer res = UInt(Sat(element1 + element2, esize, unsigned));
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

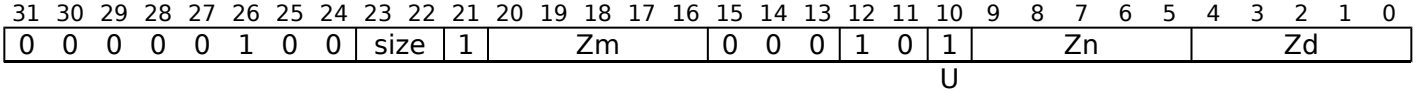
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQADD (vectors, unpredicated)

Unsigned saturating add vectors (unpredicated)

Unsigned saturating add all elements of the second source vector to corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. This instruction is unpredicated.



UQADD <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean unsigned = TRUE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + element2, esize, unsigned);

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQDECB

Unsigned saturating decrement scalar by multiple of 8-bit predicate constraint element count

Determines the number of active 8-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

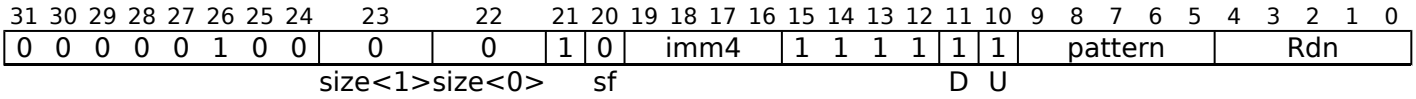
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

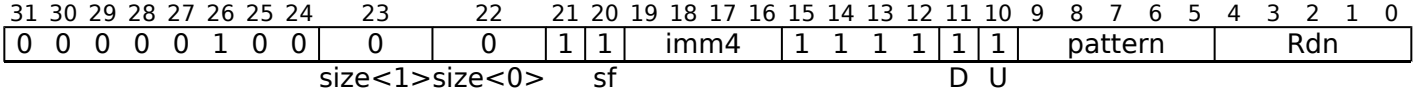
32-bit



UQDECB <Wdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;
```

64-bit



UQDECB <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;
```

Assembler Symbols

<Wdn>	Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Xdn>	Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<pattern>	Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQDECD (scalar)

Unsigned saturating decrement scalar by multiple of 64-bit predicate constraint element count

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

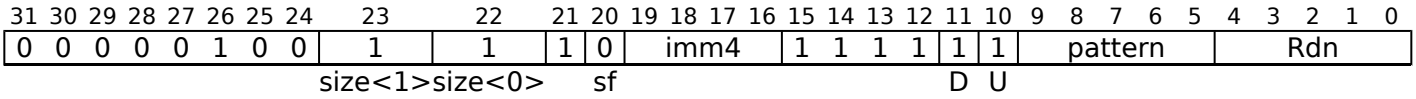
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

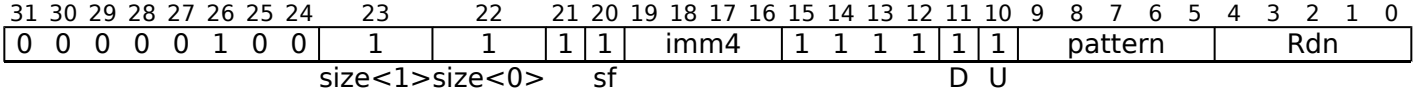
32-bit



UQDECD <Wdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;
```

64-bit



UQDECD <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;
```

Assembler Symbols

<Wdn>	Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Xdn>	Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<pattern>	Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQDECD (vector)

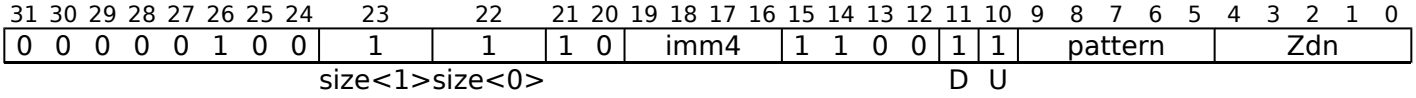
Unsigned saturating decrement vector by multiple of 64-bit predicate constraint element count

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 64-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.



UQDECD <Zdn>.D{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQDECH (scalar)

Unsigned saturating decrement scalar by multiple of 16-bit predicate constraint element count

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

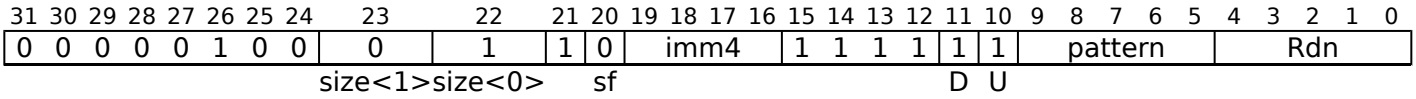
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

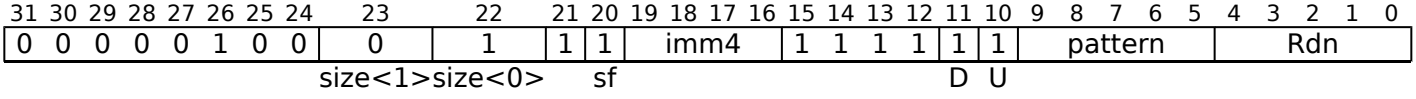
32-bit



UQDECH <Wdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;
```

64-bit



UQDECH <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;
```

Assembler Symbols

<Wdn>	Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Xdn>	Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<pattern>	Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQDECH (vector)

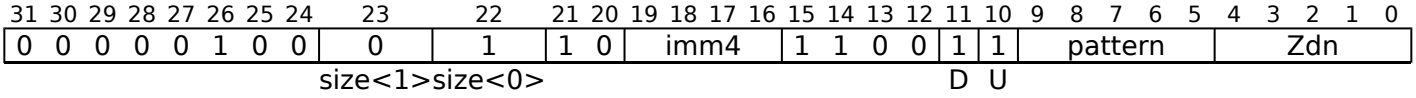
Unsigned saturating decrement vector by multiple of 16-bit predicate constraint element count

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 16-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.



UQDECH <Zdn>.H{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

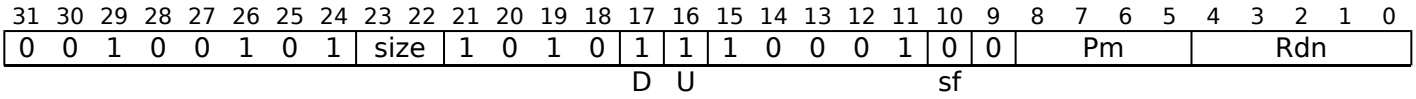
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQDECP (scalar)

Unsigned saturating decrement scalar by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.
It has encodings from 2 classes: 32-bit and 64-bit

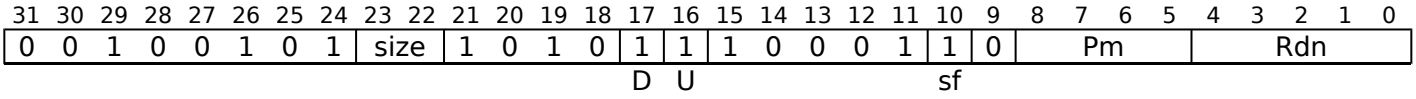
32-bit



UQDECP <Wdn>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = TRUE;
integer ssize = 32;
```

64-bit



UQDECP <Xdn>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = TRUE;
integer ssize = 64;
```

Assembler Symbols

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(ssize) operand1 = X[dn, ssize];
bits(PL) operand2 = P[m, PL];
bits(ssize) result;
integer count = 0;

for e = 0 to elements-1
    if ElemP[operand2, e, esize] == '1' then
        count = count + 1;

integer element = Int(operand1, unsigned);
(result, -) = SatQ(element - count, ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
```

Operational information

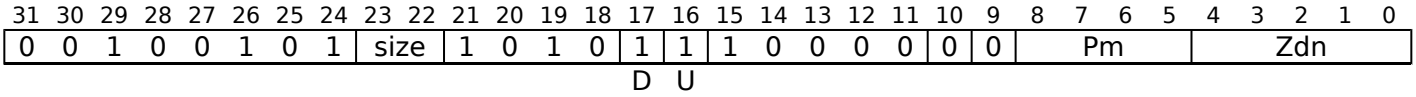
If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQDECP (vector)

Unsigned saturating decrement vector by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to decrement all destination vector elements. The results are saturated to the element unsigned integer range.
The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.



UQDECP <Zdn>.<T>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) operand2 = P[m, PL];
bits(VL) result;
integer count = 0;

for e = 0 to elements-1
    if ElemP[operand2, e, esize] == '1' then
        count = count + 1;

for e = 0 to elements-1
    integer element = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element - count, esize, unsigned);

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQDECW (scalar)

Unsigned saturating decrement scalar by multiple of 32-bit predicate constraint element count

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

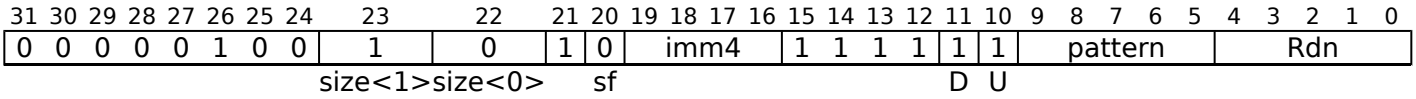
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

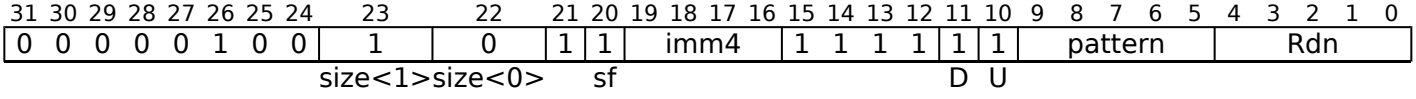
32-bit



UQDECW <Wdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;
```

64-bit



UQDECW <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;
```

Assembler Symbols

<Wdn>	Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Xdn>	Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<pattern>	Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 - (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQDECW (vector)

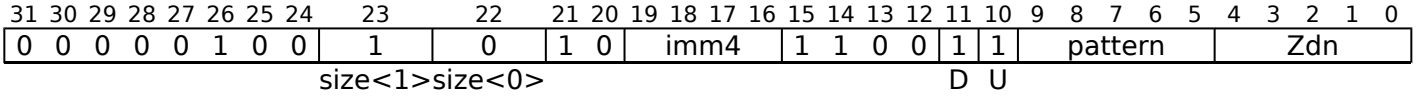
Unsigned saturating decrement vector by multiple of 32-bit predicate constraint element count

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to decrement all destination vector elements. The results are saturated to the 32-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.



UQDECW <Zdn>.S{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - (count * imm), esize, unsigned);

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQINCB

Unsigned saturating increment scalar by multiple of 8-bit predicate constraint element count

Determines the number of active 8-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

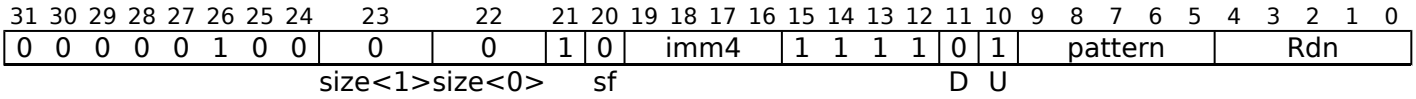
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

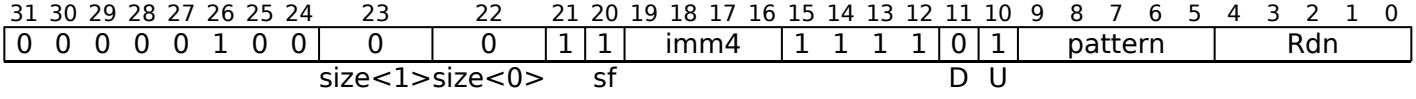
32-bit



UQINCB <Wdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;
```

64-bit



UQINCB <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;
```

Assembler Symbols

<Wdn>	Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Xdn>	Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<pattern>	Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQINCD (scalar)

Unsigned saturating increment scalar by multiple of 64-bit predicate constraint element count

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

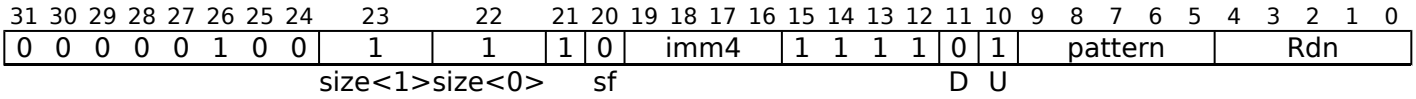
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

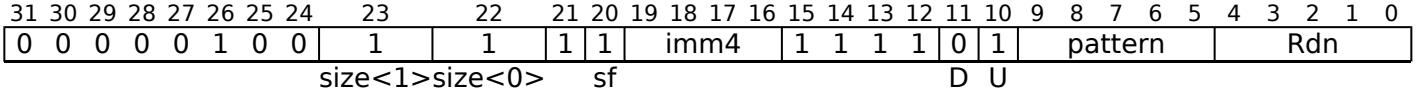
32-bit



UQINCD <Wdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;
```

64-bit



UQINCD <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;
```

Assembler Symbols

<Wdn>	Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Xdn>	Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<pattern>	Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQINCD (vector)

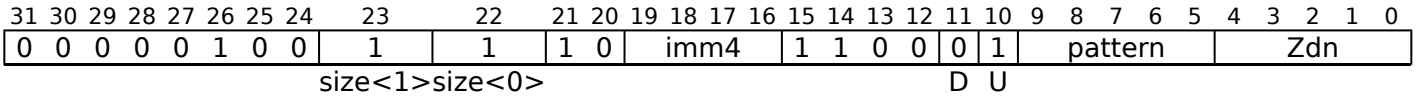
Unsigned saturating increment vector by multiple of 64-bit predicate constraint element count

Determines the number of active 64-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 64-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.



UQINCD <Zdn>.D{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 64;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + (count * imm), esize, unsigned);

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQINCH (scalar)

Unsigned saturating increment scalar by multiple of 16-bit predicate constraint element count

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

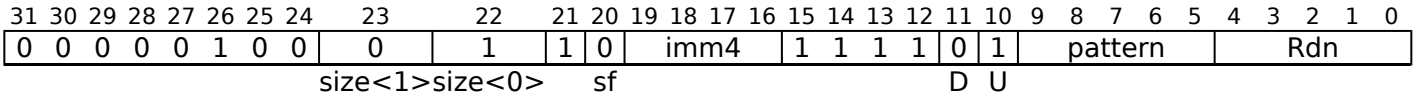
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

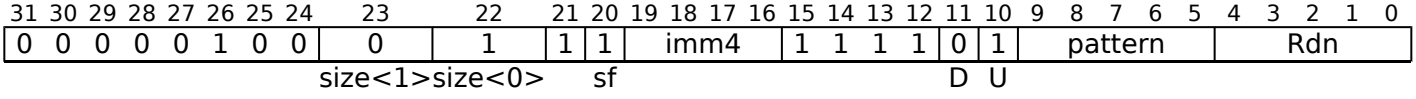
32-bit



UQINCH <Wdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;
```

64-bit



UQINCH <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;
```

Assembler Symbols

<Wdn>	Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Xdn>	Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<pattern>	Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQINCH (vector)

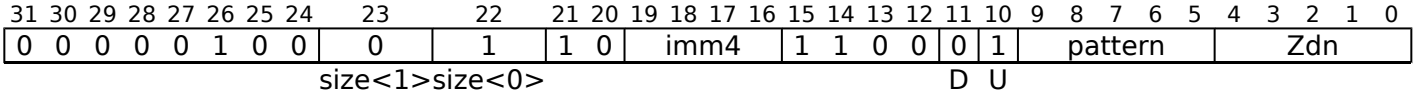
Unsigned saturating increment vector by multiple of 16-bit predicate constraint element count

Determines the number of active 16-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 16-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.



UQINCH <Zdn>.H{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + (count * imm), esize, unsigned);

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

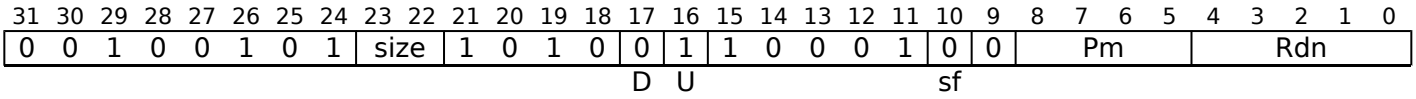
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQINCP (scalar)

Unsigned saturating increment scalar by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to increment the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.
It has encodings from 2 classes: 32-bit and 64-bit

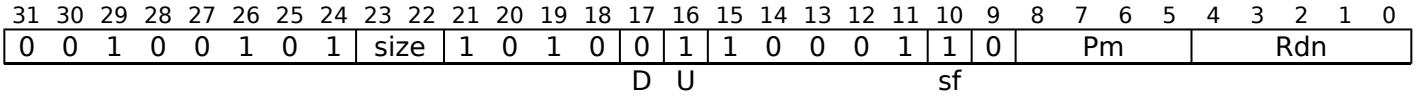
32-bit



UQINCP <Wdn>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = TRUE;
integer ssize = 32;
```

64-bit



UQINCP <Xdn>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Rdn);
boolean unsigned = TRUE;
integer ssize = 64;
```

Assembler Symbols

<Wdn> Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<Xdn> Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.

<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(ssize) operand1 = X[dn, ssize];
bits(PL) operand2 = P[m, PL];
bits(ssize) result;
integer count = 0;

for e = 0 to elements-1
    if ElemP[operand2, e, esize] == '1' then
        count = count + 1;

integer element = Int(operand1, unsigned);
(result, -) = SatQ(element + count, ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
```

Operational information

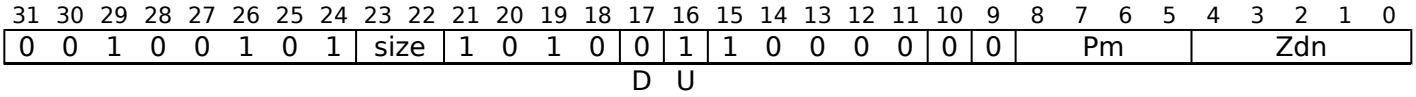
If FEAT_SME is implemented and the PE is in Streaming SVE mode, then any subsequent instruction which is dependent on the general-purpose register written by this instruction might be significantly delayed.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQINCP (vector)

Unsigned saturating increment vector by count of true predicate elements

Counts the number of true elements in the source predicate and then uses the result to increment all destination vector elements. The results are saturated to the element unsigned integer range.
The predicate size specifier may be omitted in assembler source code, but this is deprecated and will be prohibited in a future release of the architecture.



UQINCP <Zdn>.<T>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer m = UInt(Pm);
integer dn = UInt(Zdn);
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Pm> Is the name of the source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) operand2 = P[m, PL];
bits(VL) result;
integer count = 0;

for e = 0 to elements-1
    if ElemP[operand2, e, esize] == '1' then
        count = count + 1;

for e = 0 to elements-1
    integer element = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element + count, esize, unsigned);

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQINCW (scalar)

Unsigned saturating increment scalar by multiple of 32-bit predicate constraint element count

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment the scalar destination. The result is saturated to the general-purpose register's unsigned integer range.

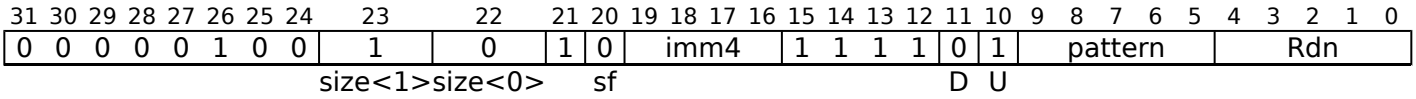
The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.

It has encodings from 2 classes: 32-bit and 64-bit

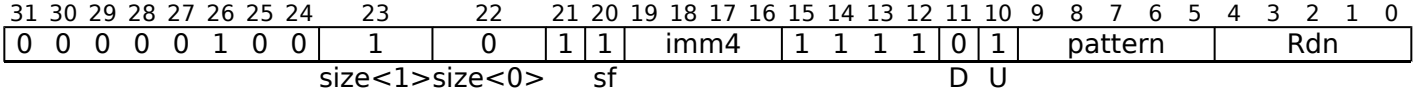
32-bit



UQINCW <Wdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 32;
```

64-bit



UQINCW <Xdn>{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Rdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
integer ssize = 64;
```

Assembler Symbols

<Wdn>	Is the 32-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<Xdn>	Is the 64-bit name of the source and destination general-purpose register, encoded in the "Rdn" field.
<pattern>	Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
integer count = DecodePredCount(pat, esize);
bits(ssize) operand1 = X[dn, ssize];
bits(ssize) result;

integer element1 = Int(operand1, unsigned);
(result, -) = SatQ(element1 + (count * imm), ssize, unsigned);
X[dn, 64] = Extend(result, 64, unsigned);
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQINCW (vector)

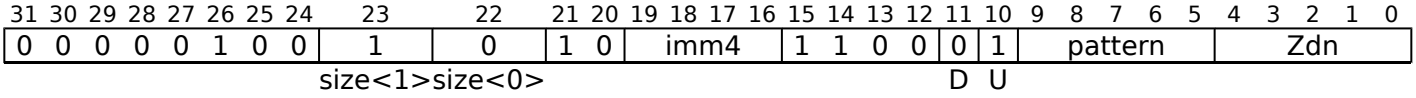
Unsigned saturating increment vector by multiple of 32-bit predicate constraint element count

Determines the number of active 32-bit elements implied by the named predicate constraint, multiplies that by an immediate in the range 1 to 16 inclusive, and then uses the result to increment all destination vector elements. The results are saturated to the 32-bit unsigned integer range.

The named predicate constraint limits the number of active elements in a single predicate to:

- A fixed number (VL1 to VL256)
- The largest power of two (POW2)
- The largest multiple of three or four (MUL3 or MUL4)
- All available, implicitly a multiple of two (ALL).

Unspecified or out of range constraint encodings generate an empty predicate or zero element count rather than Undefined Instruction exception.



UQINCW <Zdn>.S{, <pattern>{, MUL #<imm>}}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer dn = UInt(Zdn);
bits(5) pat = pattern;
integer imm = UInt(imm4) + 1;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<pattern> Is the optional pattern specifier, defaulting to ALL, encoded in “pattern”:

pattern	<pattern>
00000	POW2
00001	VL1
00010	VL2
00011	VL3
00100	VL4
00101	VL5
00110	VL6
00111	VL7
01000	VL8
01001	VL16
01010	VL32
01011	VL64
01100	VL128
01101	VL256
0111x	#uimm5
101x1	#uimm5
10110	#uimm5
1x0x1	#uimm5
1x010	#uimm5
1xx00	#uimm5
11101	MUL4
11110	MUL3
11111	ALL

<imm> Is the immediate multiplier, in the range 1 to 16, defaulting to 1, encoded in the "imm4" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
integer count = DecodePredCount(pat, esize);
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 + (count * imm), esize, unsigned);

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

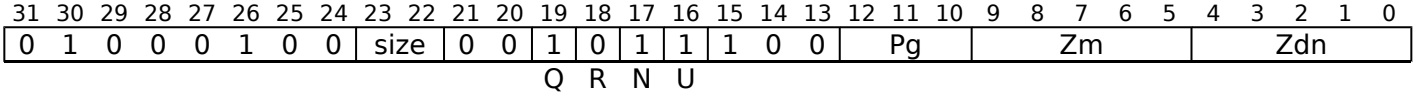
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQRSHL

Unsigned saturating rounding shift left by vector (predicated)

Shift active unsigned elements of the first source vector by corresponding elements of the second source vector and destructively place the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. Inactive elements in the destination vector register remain unmodified.



UQRSHL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element = UInt(Elem[operand1, e, esize]);
        integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
        integer round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
        integer res = (element + round_const) << shift;
        Elem[result, e, esize] = UnsignedSat(res, esize);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

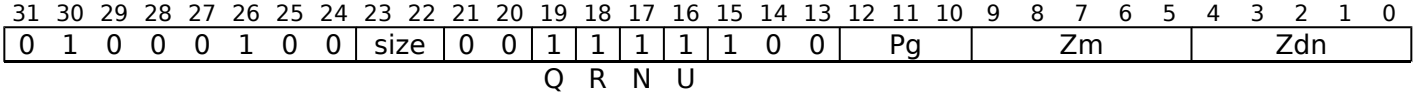
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQRSHLR

Unsigned saturating rounding shift left reversed vectors (predicated)

Shift active unsigned elements of the second source vector by corresponding elements of the first source vector and destructively place the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. Inactive elements in the destination vector register remain unmodified.



UQRSHLR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand2 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element = UInt(Elem[operand1, e, esize]);
        integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
        integer round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
        integer res = (element + round_const) << shift;
        Elem[result, e, esize] = UnsignedSat(res, esize);
    else
        Elem[result, e, esize] = Elem[operand2, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

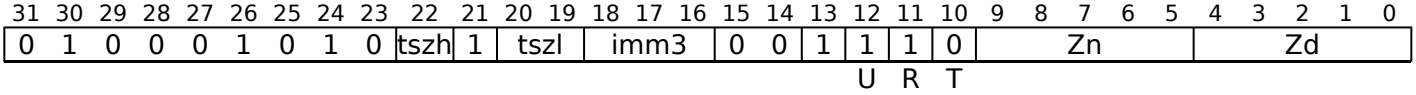
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQRSHRNB

Unsigned saturating rounding shift right narrow by immediate (bottom)

Shift each unsigned integer value in the source vector elements right by an immediate value, and place the rounded results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



UQRSHRNB <Zd>.<T>, <Zn>.<Tb>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '000' UNDEFINED;
    when '001' esize = 8;
    when '01x' esize = 16;
    when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

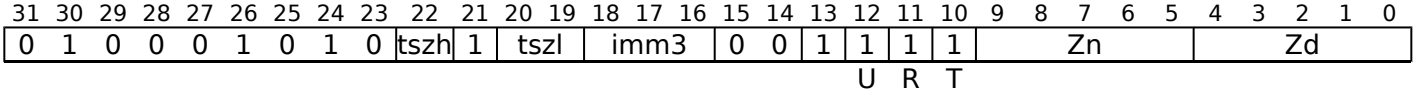
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;
integer round_const = 1 << (shift-1);

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = (UInt(element) + round_const) >> shift;
    Elem[result, 2*e + 0, esize] = UnsignedSat(res, esize);
    Elem[result, 2*e + 1, esize] = Zeros(esize);

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQRSHRNT

Unsigned saturating rounding shift right narrow by immediate (top)

Shift each unsigned integer value in the source vector elements by an immediate value, and place the rounded results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



UQRSHRNT <Zd>.<T>, <Zn>.<Tb>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '000' UNDEFINED;
    when '001' esize = 8;
    when '01x' esize = 16;
    when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

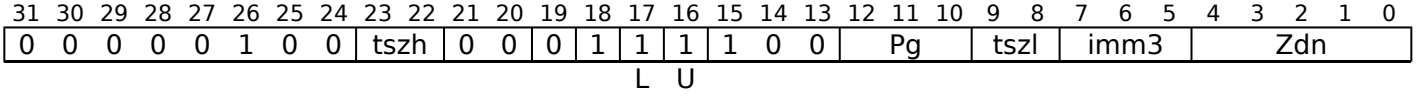
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result = Z[d, VL];
integer round_const = 1 << (shift-1);

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = (UInt(element) + round_const) >> shift;
    Elem[result, 2*e + 1, esize] = UnsignedSat(res, esize);

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQSHL (immediate)

Unsigned saturating shift left by immediate

Shift left by immediate each active unsigned element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. Inactive elements in the destination vector register remain unmodified.



UQSHL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
case tsize of
    when '0000' UNDEFINED;
    when '0001' esize = 8;
    when '001x' esize = 16;
    when '01xx' esize = 32;
    when '1xxx' esize = 64;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = UInt(tsize:imm3) - esize;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) mask = P[g, PL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    if ElemP[mask, e, esize] == '1' then
        integer res = element1 << shift;
        Elem[result, e, esize] = UnsignedSat(res, esize);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

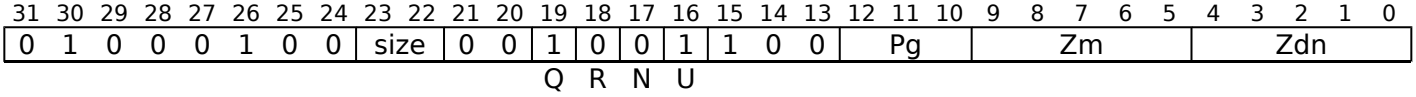
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQSHL (vectors)

Unsigned saturating shift left by vector (predicated)

Shift active unsigned elements of the first source vector by corresponding elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. Inactive elements in the destination vector register remain unmodified.



UQSHL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element = UInt(Elem[operand1, e, esize]);
        integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
        integer res = element << shift;
        Elem[result, e, esize] = UnsignedSat(res, esize);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

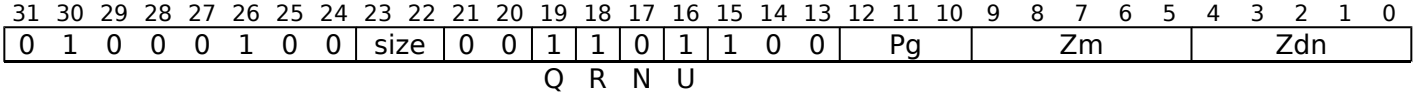
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQSHLR

Unsigned saturating shift left reversed vectors (predicated)

Shift active unsigned elements of the second source vector by corresponding elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. Inactive elements in the destination vector register remain unmodified.



UQSHLR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand2 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element = UInt(Elem[operand1, e, esize]);
        integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
        integer res = element << shift;
        Elem[result, e, esize] = UnsignedSat(res, esize);
    else
        Elem[result, e, esize] = Elem[operand2, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

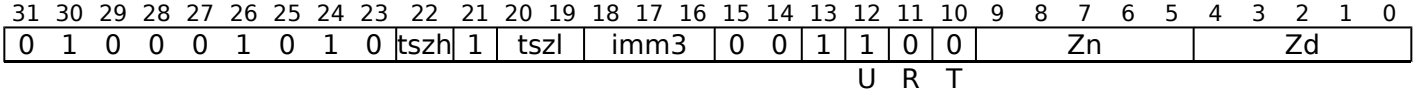
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQSHRNB

Unsigned saturating shift right narrow by immediate (bottom)

Shift each unsigned integer value in the source vector elements right by an immediate value, and place the truncated results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



UQSHRNB <Zd>.<T>, <Zn>.<Tb>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '000' UNDEFINED;
    when '001' esize = 8;
    when '01x' esize = 16;
    when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

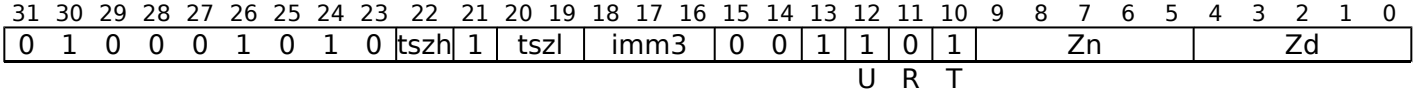
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = UInt(element) >> shift;
    Elem[result, 2*e + 0, esize] = UnsignedSat(res, esize);
    Elem[result, 2*e + 1, esize] = Zeros(esize);

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQSHRNT

Unsigned saturating shift right narrow by immediate (top)

Shift each unsigned integer value in the source vector elements right by an immediate value, and place the truncated results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged. Each result element is saturated to the half-width N-bit element's unsigned integer range 0 to (2^N)-1. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



UQSHRNT <Zd>.<T>, <Zn>.<Tb>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '000' UNDEFINED;
    when '001' esize = 8;
    when '01x' esize = 16;
    when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

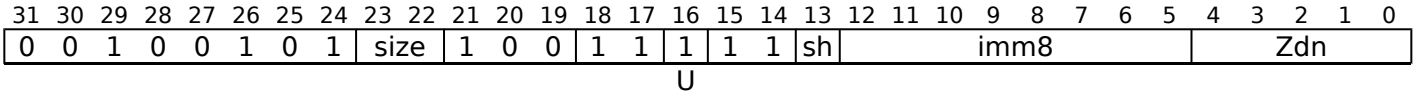
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    bits(2*esize) element = Elem[operand, e, 2*esize];
    integer res = UInt(element) >> shift;
    Elem[result, 2*e + 1, esize] = UnsignedSat(res, esize);

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQSUB (immediate)

Unsigned saturating subtract immediate (unpredicated)

Unsigned saturating subtract an unsigned immediate from each element of the source vector, and destructively place the results in the corresponding elements of the source vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. This instruction is unpredicated.
The immediate is an unsigned value in the range 0 to 255, and for element widths of 16 bits or higher it may also be a positive multiple of 256 in the range 256 to 65280.
The immediate is encoded in 8 bits with an optional left shift by 8. The preferred disassembly when the shift option is specified is "#<uimm8>, LSL #8". However an assembler and disassembler may also allow use of the shifted 16-bit value unless the immediate is 0 and the shift amount is 8, which must be unambiguously described as "#0, LSL #8".



UQSUB <Zdn>.<T>, <Zdn>.<T>, #<imm>{, <shift>}

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size:sh == '001' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer dn = UInt(Zdn);
integer imm = UInt(imm8);
if sh == '1' then imm = imm << 8;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<imm> Is an unsigned immediate in the range 0 to 255, encoded in the "imm8" field.

<shift> Is the optional left shift to apply to the immediate, defaulting to LSL #0 and encoded in “sh”:

sh	<shift>
0	LSL #0
1	LSL #8

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - imm, esize, unsigned);

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

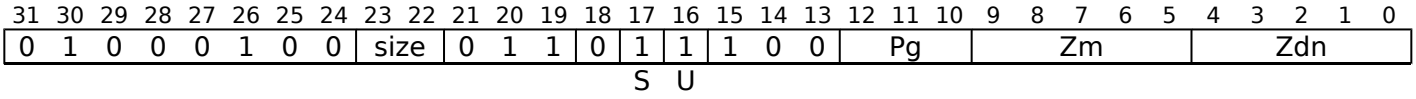
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQSUB (vectors, predicated)

Unsigned saturating subtraction (predicated)

Subtract active unsigned elements of the second source vector from corresponding unsigned elements of the first source vector and destructively place the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. Inactive elements in the destination vector register remain unmodified.



UQSUB <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    if ElemP[mask, e, esize] == '1' then
        integer res = UInt(Sat(element1 - element2, esize, unsigned));
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

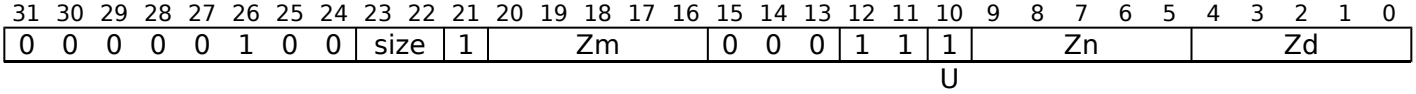
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQSUB (vectors, unpredicated)

Unsigned saturating subtract vectors (unpredicated)

Unsigned saturating subtract all elements of the second source vector from corresponding elements of the first source vector and place the results in the corresponding elements of the destination vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. This instruction is unpredicated.



UQSUB <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
boolean unsigned = TRUE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

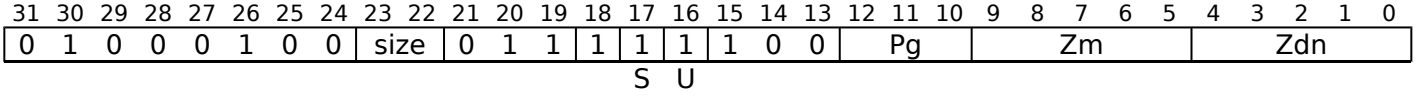
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, e, esize], unsigned);
    integer element2 = Int(Elem[operand2, e, esize], unsigned);
    (Elem[result, e, esize], -) = SatQ(element1 - element2, esize, unsigned);

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQSUBR

Unsigned saturating subtraction reversed vectors (predicated)

Subtract active unsigned elements of the first source vector from corresponding unsigned elements of the second source vector and destructively place the results in the corresponding elements of the first source vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. Inactive elements in the destination vector register remain unmodified.



UQSUBR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
boolean unsigned = TRUE;
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    if ElemP[mask, e, esize] == '1' then
        integer res = UInt(Sat(element2 - element1, esize, unsigned));
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

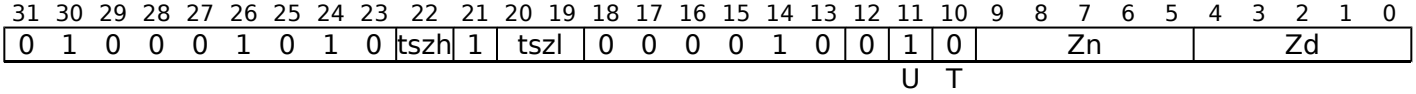
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQXTNB

Unsigned saturating extract narrow (bottom)

Saturate the unsigned integer value in each source element to half the original source element width, and place the results in the even-numbered half-width destination elements, while setting the odd-numbered elements to zero.



UQXTNB <Zd>.<T>, <Zn>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '001' esize = 16;
    when '010' esize = 32;
    when '100' esize = 64;
    otherwise UNDEFINED;
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	10	H
x	11	RESERVED
1	00	S
1	01	RESERVED
1	10	RESERVED

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	10	S
x	11	RESERVED
1	00	D
1	01	RESERVED
1	10	RESERVED

Operation

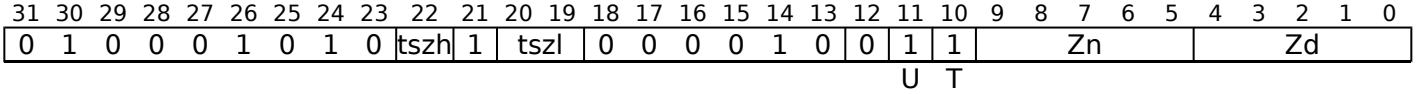
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) result;
constant integer halfesize = esize DIV 2;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    bits(halfesize) res = UnsignedSat(element1, halfesize);
    Elem[result, 2*e + 0, halfesize] = res;
    Elem[result, 2*e + 1, halfesize] = Zeros(halfesize);

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UQXTNT

Unsigned saturating extract narrow (top)

Saturate the unsigned integer value in each source element to half the original source element width, and place the results in the odd-numbered half-width destination elements, leaving the even-numbered elements unchanged.



UQXTNT <Zd>.<T>, <Zn>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '001' esize = 16;
    when '010' esize = 32;
    when '100' esize = 64;
    otherwise UNDEFINED;
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	B
0	10	H
x	11	RESERVED
1	00	S
1	01	RESERVED
1	10	RESERVED

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	H
0	10	S
x	11	RESERVED
1	00	D
1	01	RESERVED
1	10	RESERVED

Operation

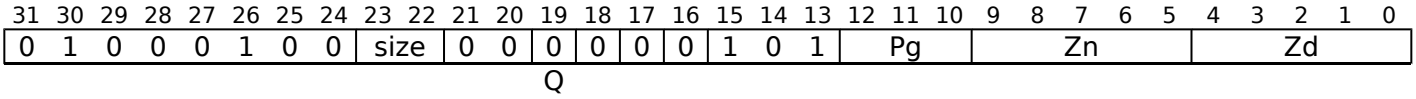
```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) result = Z[d, VL];
constant integer halfesize = esize DIV 2;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    bits(halfesize) res = UnsignedSat(element1, halfesize);
    Elem[result, 2*e + 1, halfesize] = res;

Z[d, VL] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

URECPE

Unsigned reciprocal estimate (predicated)

Find the approximate reciprocal of each active unsigned element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.



URECPE <Zd>.S, <Pg>/M, <Zn>.S

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size != '10' then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = UnsignedRecipEstimate(element);

Z[d, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

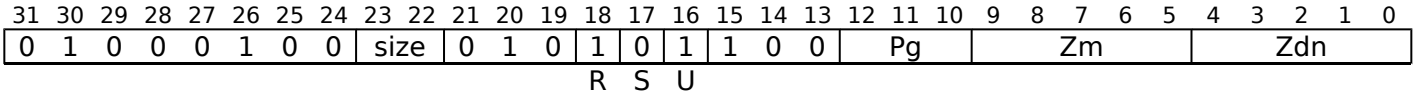
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

URHADD

Unsigned rounding halving addition

Add active unsigned elements of the first source vector to corresponding unsigned elements of the second source vector, shift right one bit, and destructively place the rounded results in the corresponding elements of the first source vector. Inactive elements in the destination vector register remain unmodified.



URHADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;
integer round_const = 1;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, e, esize]);
    if ElemP[mask, e, esize] == '1' then
        integer res = (element1 + element2 + round_const) >> 1;
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

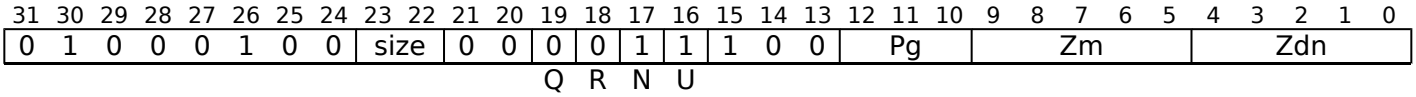
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

URSHL

Unsigned rounding shift left by vector (predicated)

Shift active unsigned elements of the first source vector by corresponding elements of the second source vector and destructively place the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Inactive elements in the destination vector register remain unmodified.



URSHL <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element = UInt(Elem[operand1, e, esize]);
        integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
        integer round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
        integer res = (element + round_const) << shift;
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

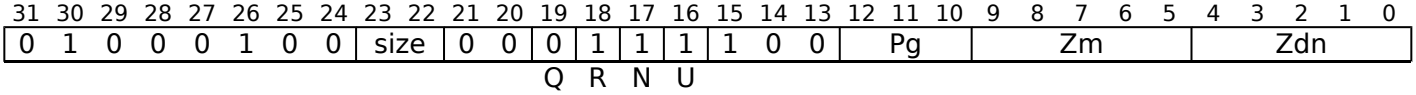
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

URSHLR

Unsigned rounding shift left reversed vectors (predicated)

Shift active unsigned elements of the second source vector by corresponding elements of the first source vector and destructively place the rounded results in the corresponding elements of the first source vector. A positive shift amount performs a left shift, otherwise a right shift by the negated shift amount is performed. Inactive elements in the destination vector register remain unmodified.



URSHLR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer m = UInt(Zm);
integer dn = UInt(Zdn);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) operand2 = Z[dn, VL];
bits(VL) result;

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        integer element = UInt(Elem[operand1, e, esize]);
        integer shift = ShiftSat(SInt(Elem[operand2, e, esize]), esize);
        integer round_const = 1 << (-shift - 1); // 0 for left shift, 2^(n-1) for right shift
        integer res = (element + round_const) << shift;
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand2, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

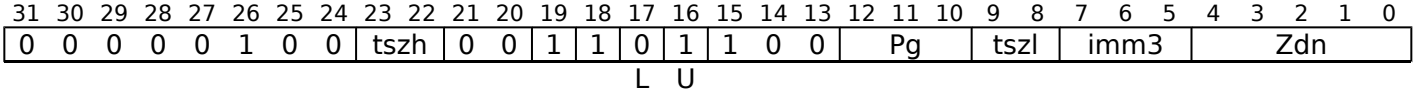
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

URSHR

Unsigned rounding shift right by immediate

Shift right by immediate each active unsigned element of the source vector, and destructively place the rounded results in the corresponding elements of the source vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. Inactive elements in the destination vector register remain unmodified.



URSHR <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
case tsize of
    when '0000' UNDEFINED;
    when '0001' esize = 8;
    when '001x' esize = 16;
    when '01xx' esize = 32;
    when '1xxx' esize = 64;
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zdn> Is the name of the source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(PL) mask = P[g, PL];
bits(VL) result;
integer round_const = 1 << (shift-1);

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    if ElemP[mask, e, esize] == '1' then
        integer res = (element1 + round_const) >> shift;
        Elem[result, e, esize] = res<esize-1:0>;
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

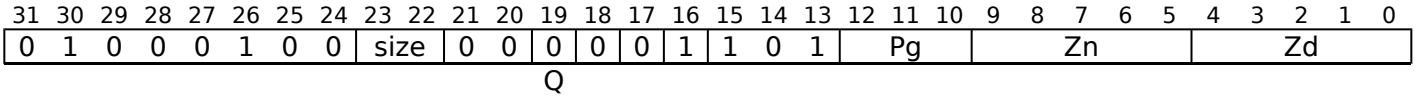
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

URSQRTE

Unsigned reciprocal square root estimate (predicated)

Find the approximate reciprocal square root of each active unsigned element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.



URSQRTE <Zd>.S, <Pg>/M, <Zn>.S

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size != '10' then UNDEFINED;
constant integer esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd>	Is the name of the destination scalable vector register, encoded in the "Zd" field.
<Pg>	Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.
<Zn>	Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = UnsignedRSqrtEstimate(element);

Z[d, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

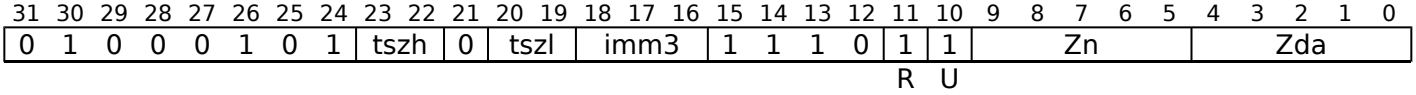
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

URSRA

Unsigned rounding shift right and accumulate (immediate)

Shift right by immediate each unsigned element of the source vector, inserting zeroes, and add the rounded intermediate result destructively to the corresponding elements of the addend vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



URSRA <Zda>.<T>, <Zn>.<T>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
case tsize of
    when '0000' UNDEFINED;
    when '0001' esize = 8;
    when '001x' esize = 16;
    when '01xx' esize = 32;
    when '1xxx' esize = 64;
integer n = UInt(Zn);
integer da = UInt(Zda);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[da, VL];
bits(VL) result;
integer round_const = 1 << (shift - 1);

for e = 0 to elements-1
    integer element = (UInt(Elem[operand1, e, esize]) + round_const) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

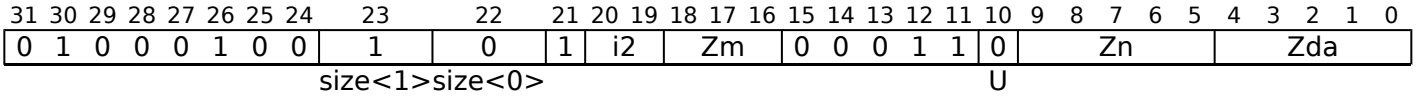
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

USDOT (indexed)

Unsigned by signed integer indexed dot product

The unsigned by signed integer indexed dot product instruction computes the dot product of a group of four unsigned 8-bit integer values held in each 32-bit element of the first source vector multiplied by a group of four signed 8-bit integer values in an indexed 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.
The groups within the second source vector are specified using an immediate index which selects the same group position within each 128-bit vector segment. The index range is from 0 to 3. This instruction is unpredicated.
ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

SVE
(FEAT_I8MM)



USDOT <Zda>.S, <Zn>.B, <Zm>.B[<imm>]

```
if (!HaveSVE() && !HaveSME()) || !HaveInt8MatMulExt() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.
<imm>	Is the immediate index of a 32-bit group of four 8-bit values within each 128-bit vector segment, in the range 0 to 3, encoded in the "i2" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer eltspersegment = 128 DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer segmentbase = e - (e MOD eltspersegment);
    integer s = segmentbase + index;
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 3
        integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        integer element2 = SInt(Elem[operand2, 4 * s + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = res;

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

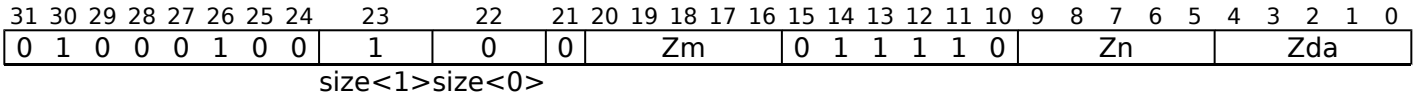
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

USDOT (vectors)

Unsigned by signed integer dot product

The unsigned by signed integer dot product instruction computes the dot product of a group of four unsigned 8-bit integer values held in each 32-bit element of the first source vector multiplied by a group of four signed 8-bit integer values in the corresponding 32-bit element of the second source vector, and then destructively adds the widened dot product to the corresponding 32-bit element of the destination vector.
This instruction is unpredicated.
ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

SVE
(FEAT_I8MM)



USDOT <Zda>.S, <Zn>.B, <Zm>.B

```
if (!HaveSVE() && !HaveSME()) || !HaveInt8MatMulExt() then UNDEFINED;
constant integer esize = 32;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) res = Elem[operand3, e, esize];
    for i = 0 to 3
        integer element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
        integer element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
        res = res + element1 * element2;
    Elem[result, e, esize] = res;

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.

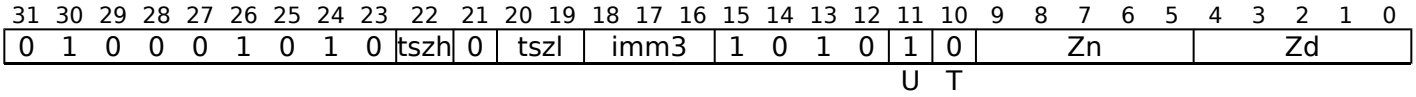
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

USHLLB

Unsigned shift left long by immediate (bottom)

Shift left by immediate each even-numbered unsigned element of the source vector, and place the results in the overlapping double-width elements of the destination vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. This instruction is unpredicated.



USHLLB <Zd>.<T>, <Zn>.<Tb>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '000' UNDEFINED;
    when '001' esize = 8;
    when '01x' esize = 16;
    when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = UInt(tsize:imm3) - esize;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element = Elem[operand, 2*e + 0, esize];
    integer shifted_value = UInt(element) << shift;
    Elem[result, e, 2*esize] = shifted_value<2*esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

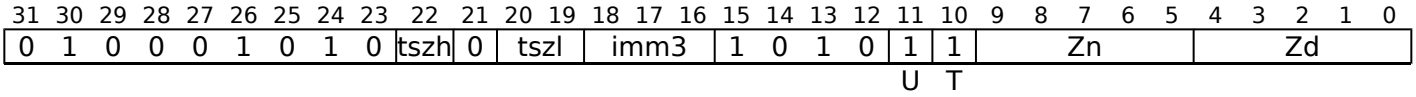
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

USHLLT

Unsigned shift left long by immediate (top)

Shift left by immediate each odd-numbered unsigned element of the source vector, and place the results in the overlapping double-width elements of the destination vector. The immediate shift amount is an unsigned value in the range 0 to number of bits per element minus 1. This instruction is unpredicated.



USHLLT <Zd>.<T>, <Zn>.<Tb>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(3) tsize = tszh:tszl;
integer esize;
case tsize of
    when '000' UNDEFINED;
    when '001' esize = 8;
    when '01x' esize = 16;
    when '1xx' esize = 32;
integer n = UInt(Zn);
integer d = UInt(Zd);
integer shift = UInt(tsize:imm3) - esize;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
0	00	RESERVED
0	01	H
0	1x	S
1	xx	D

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<Tb>
0	00	RESERVED
0	01	B
0	1x	H
1	xx	S

<const> Is the immediate shift amount, in the range 0 to number of bits per element minus 1, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV (2 * esize);
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element = Elem[operand, 2*e + 1, esize];
    integer shifted_value = UInt(element) << shift;
    Elem[result, e, 2*esize] = shifted_value<2*esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

USMMLA

Unsigned by signed integer matrix multiply-accumulate

The unsigned by signed integer matrix multiply-accumulate instruction multiplies the 2×8 matrix of unsigned 8-bit integer values held in each 128-bit segment of the first source vector by the 8×2 matrix of signed 8-bit integer values in the corresponding segment of the second source vector. The resulting 2×2 widened 32-bit integer matrix product is then destructively added to the 32-bit integer matrix accumulator held in the corresponding segment of the addend and destination vector. This is equivalent to performing an 8-way dot product per destination element.

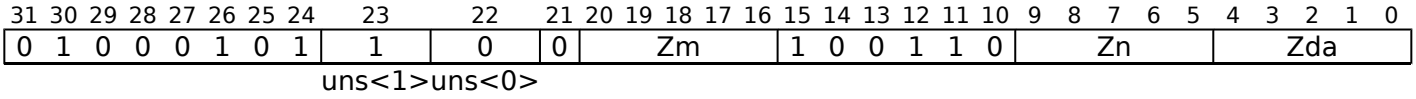
This instruction is unpredicated.

ID_AA64ZFR0_EL1.I8MM indicates whether this instruction is implemented.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

SVE

(FEAT_I8MM)



USMMLA <Zda>.S, <Zn>.B, <Zm>.B

```
if !HaveSVE() || !HaveInt8MatMulExt() then UNDEFINED;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
boolean op1_unsigned = TRUE;
boolean op2_unsigned = FALSE;
```

Assembler Symbols

<Zda>	Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer segments = VL DIV 128;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result = Zeros(VL);
bits(128) op1, op2;
bits(128) res, addend;

for s = 0 to segments-1
    op1    = Elem[operand1, s, 128];
    op2    = Elem[operand2, s, 128];
    addend = Elem[operand3, s, 128];
    res    = MatMulAdd(addend, op1, op2, op1_unsigned, op2_unsigned);
    Elem[result, s, 128] = res;

Z[da, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

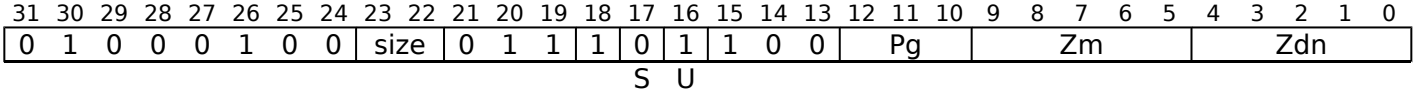
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

USQADD

Unsigned saturating addition of signed value

Add active signed elements of the source vector to the corresponding unsigned elements of the addend vector, and destructively place the results in the corresponding elements of the addend vector. Each result element is saturated to the N-bit element's unsigned integer range 0 to (2^N)-1. Inactive elements in the destination vector register remain unmodified.



USQADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer g = UInt(Pg);
integer dn = UInt(Zdn);
integer m = UInt(Zm);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = if AnyActiveElement(mask, esize) then Z[m, VL] else Zeros(VL);
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = UnsignedSat(UInt(element1) + SInt(element2), esize);
    else
        Elem[result, e, esize] = Elem[operand1, e, esize];

Z[dn, VL] = result;
```

Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

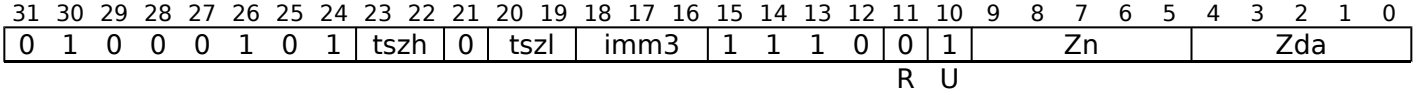
- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

USRA

Unsigned shift right and accumulate (immediate)

Shift right by immediate each unsigned element of the source vector, inserting zeroes, and add the truncated intermediate result destructively to the corresponding elements of the addend vector. The immediate shift amount is an unsigned value in the range 1 to number of bits per element. This instruction is unpredicated.



USRA <Zda>.<T>, <Zn>.<T>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
case tsize of
    when '0000' UNDEFINED;
    when '0001' esize = 8;
    when '001x' esize = 16;
    when '01xx' esize = 32;
    when '1xxx' esize = 64;
integer n = UInt(Zn);
integer da = UInt(Zda);
integer shift = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zda> Is the name of the third source and destination scalable vector register, encoded in the "Zda" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[da, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element = UInt(Elem[operand1, e, esize]) >> shift;
    Elem[result, e, esize] = Elem[operand2, e, esize] + element<esize-1:0>;

Z[da, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.

    - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

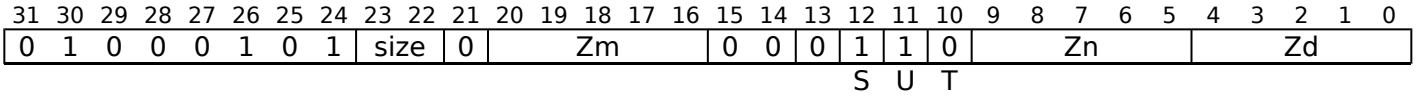
- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

USUBLB

Unsigned subtract long (bottom)

Subtract the even-numbered unsigned elements of the second source vector from the corresponding unsigned elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



USUBLB <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 0;
integer sel2 = 0;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 - element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

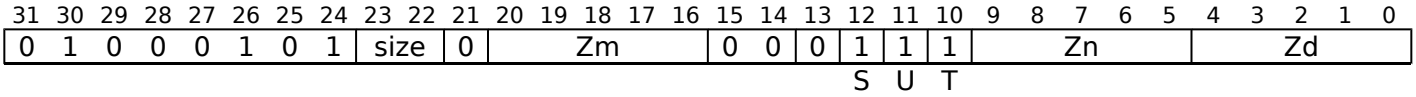
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

USUBLT

Unsigned subtract long (top)

Subtract the odd-numbered unsigned elements of the second source vector from the corresponding unsigned elements of the first source vector, and place the results in the overlapping double-width elements of the destination vector. This instruction is unpredicated.



USUBLT <Zd>.<T>, <Zn>.<Tb>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer sel1 = 1;
integer sel2 = 1;
boolean unsigned = TRUE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = Int(Elem[operand1, 2*e + sel1, esize DIV 2], unsigned);
    integer element2 = Int(Elem[operand2, 2*e + sel2, esize DIV 2], unsigned);
    integer res = element1 - element2;
    Elem[result, e, esize] = res<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

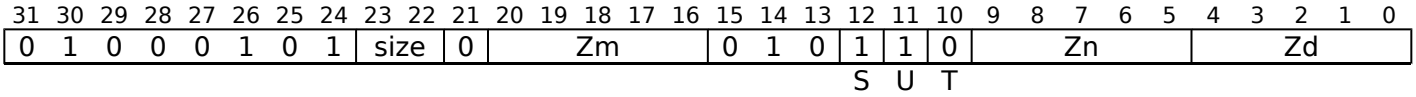
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

USUBWB

Unsigned subtract wide (bottom)

Subtract the even-numbered unsigned elements of the second source vector from the overlapping double-width elements of the first source vector and place the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated.



USUBWB <Zd>.<T>, <Zn>.<T>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, 2*e + 0, esize DIV 2]);
    Elem[result, e, esize] = (element1 - element2)<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

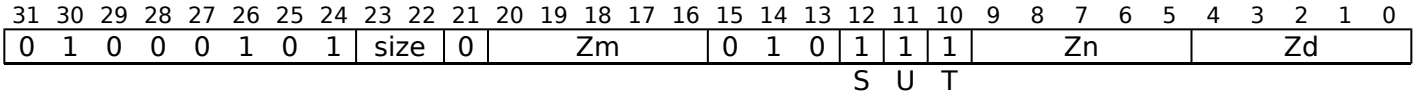
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

USUBWT

Unsigned subtract wide (top)

Subtract the odd-numbered unsigned elements of the second source vector from the overlapping double-width elements of the first source vector and place the results in the corresponding double-width elements of the destination vector. This instruction is unpredicated. This instruction is unpredicated.



USUBWT <Zd>.<T>, <Zn>.<T>, <Zm>.<Tb>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    integer element1 = UInt(Elem[operand1, e, esize]);
    integer element2 = UInt(Elem[operand2, 2*e + 1, esize DIV 2]);
    Elem[result, e, esize] = (element1 - element2)<esize-1:0>;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

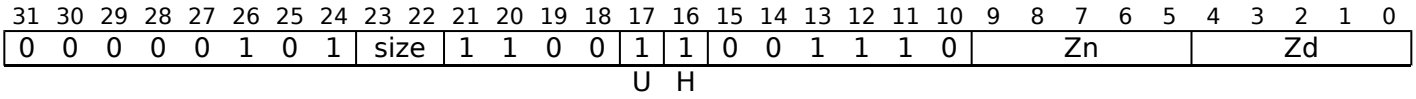
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UUNPKHI, UUNPKLO

Unsigned unpack and extend half of vector

Unpack elements from the lowest or highest half of the source vector and then zero-extend them to place in elements of twice their size within the destination vector. This instruction is unpredicated.
It has encodings from 2 classes: High half and Low half

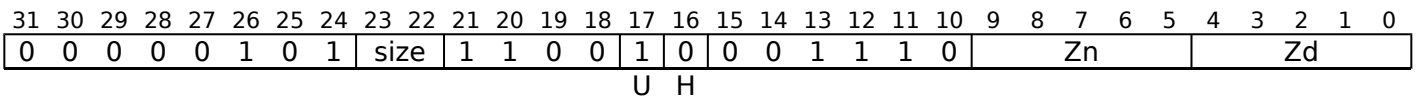
High half



UUNPKHI <Zd>.<T>, <Zn>.<Tb>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = TRUE;
boolean hi = TRUE;
```

Low half



UUNPKLO <Zd>.<T>, <Zn>.<Tb>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = TRUE;
boolean hi = FALSE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

<Tb> Is the size specifier, encoded in “size”:

size	<Tb>
00	RESERVED
01	B
10	H
11	S

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
constant integer hsize = esize DIV 2;
bits(VL) operand = Z[n, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(hsize) element = if hi then Elem[operand, e + elements, hsize] else Elem[operand, e, hsize];
    Elem[result, e, esize] = Extend(element, esize, unsigned);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

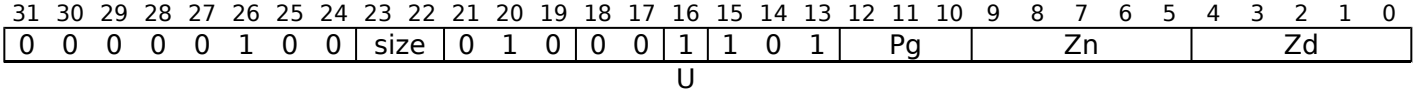
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UXTB, UXTH, UXTW

Unsigned byte / halfword / word extend (predicated)

Zero-extend the least-significant sub-element of each active element of the source vector, and place the results in the corresponding elements of the destination vector. Inactive elements in the destination vector register remain unmodified.
It has encodings from 3 classes: Byte , Halfword and Word

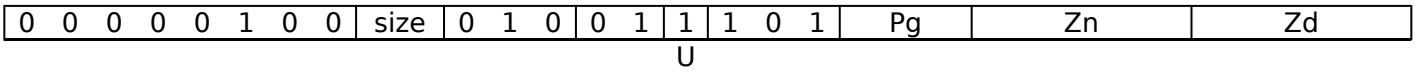
Byte



UXTB <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size == '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer s_esize = 8;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = TRUE;
```

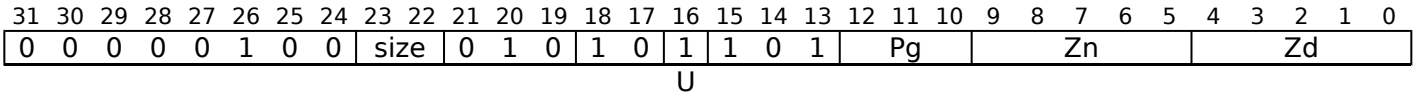
Halfword



UXTH <Zd>.<T>, <Pg>/M, <Zn>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size IN {'0x'} then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer s_esize = 16;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = TRUE;
```

Word



UXTW <Zd>.D, <Pg>/M, <Zn>.D

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
if size != '11' then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer s_esize = 32;
integer g = UInt(Pg);
integer n = UInt(Zn);
integer d = UInt(Zd);
boolean unsigned = TRUE;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> For the byte variant: is the size specifier, encoded in “size”:

size	<T>
00	RESERVED
01	H
10	S
11	D

For the halfword variant: is the size specifier, encoded in “size<0>”:

size<0>	<T>
0	S
1	D

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = if AnyActiveElement(mask, esize) then Z[n, VL] else Zeros(VL);
bits(VL) result = Z[d, VL];

for e = 0 to elements-1
    if ElemP[mask, e, esize] == '1' then
        bits(esize) element = Elem[operand, e, esize];
        Elem[result, e, esize] = Extend(element<s_esize-1:0>, esize, unsigned);

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is UNPREDICTABLE:

- The MOVPRFX instruction must be unpredicated, or be predicated using the same governing predicate register and source element size as this instruction.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

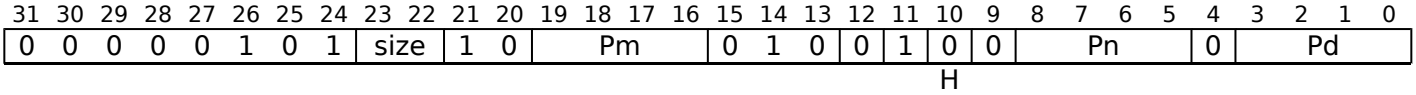
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UZP1, UZP2 (predicates)

Concatenate even or odd elements from two predicates

Concatenate adjacent even or odd-numbered elements from the first and second source predicates and place in elements of the destination predicate. This instruction is unpredicated.
It has encodings from 2 classes: Even and Odd

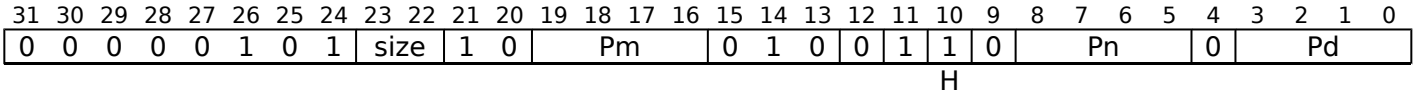
Even



UZP1 <Pd>.<T>, <Pn>.<T>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
integer part = 0;
```

Odd



UZP2 <Pd>.<T>, <Pn>.<T>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
integer part = 1;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (esize * 2);
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;

for p = 0 to pairs - 1
    Elem[result, p, esize DIV 8] = Elem[operand1, 2*p+part, esize DIV 8];

for p = 0 to pairs - 1
    Elem[result, pairs+p, esize DIV 8] = Elem[operand2, 2*p+part, esize DIV 8];

P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

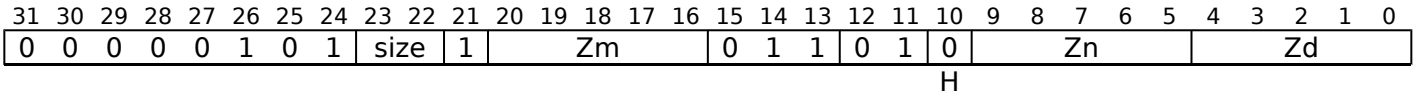
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

UZP1, UZP2 (vectors)

Concatenate even or odd elements from two vectors

Concatenate adjacent even or odd-numbered elements from the first and second source vectors and place in elements of the destination vector. This instruction is unpredicated. The 128-bit element variant of this instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits are set to zero.
ID_AA64ZFR0_EL1.F64MM indicates whether the 128-bit element variant of the instruction is implemented.
It has encodings from 4 classes: Even , Even (quadwords) , Odd and Odd (quadwords)

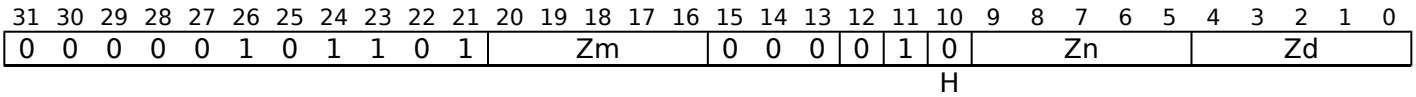
Even



UZP1 <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;
```

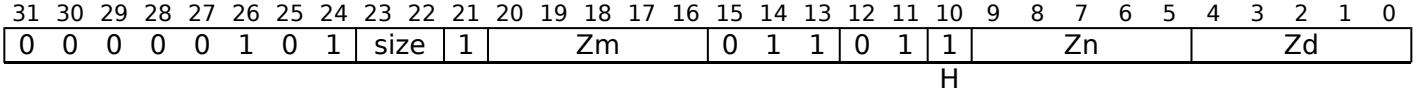
Even (quadwords)
(FEAT_F64MM)



UZP1 <Zd>.Q, <Zn>.Q, <Zm>.Q

```
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
constant integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;
```

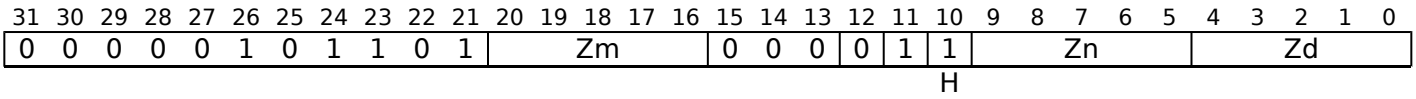
Odd



UZP2 <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;
```

Odd (quadwords)
(FEAT_F64MM)



UZP2 <Zd>.Q, <Zn>.Q, <Zm>.Q

```
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
constant integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
if esize < 128 then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
if VL < esize * 2 then UNDEFINED;
constant integer pairs = VL DIV (esize * 2);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Zeros(VL);

for p = 0 to pairs - 1
    Elem[result, p, esize] = Elem[operand1, 2*p+part, esize];

for p = 0 to pairs - 1
    Elem[result, pairs+p, esize] = Elem[operand2, 2*p+part, esize];

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

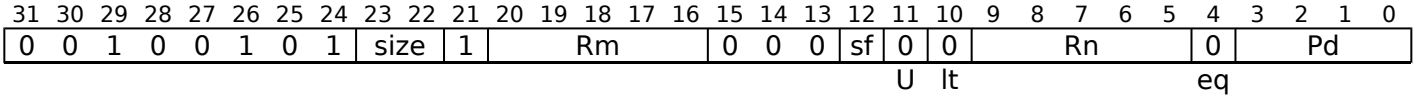
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

WHILEGE

While decrementing signed scalar greater than or equal to scalar

Generate a predicate that starting from the highest numbered element is true while the decrementing value of the first, signed scalar operand is greater than or equal to the second scalar operand and false thereafter down to the lowest numbered element.
If the second scalar operand is equal to the minimum signed integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.
The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.
The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.



WHILEGE <Pd>.<T>, <R><n>, <R><m>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = FALSE;
SVECmp op = Cmp_GE;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<R> Is a width specifier, encoded in “sf”:

sf	<R>
0	W
1	X

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;

for e = elements-1 downto 0
    boolean cond;
    case op of
        when Cmp_GT cond = (Int(operand1, unsigned) >  Int(operand2, unsigned));
        when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));

    last = last && cond;
    ElemP[result, e, esize] = if last then '1' else '0';
    operand1 = operand1 - 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

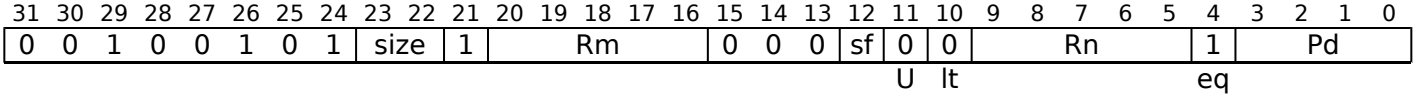
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

WHILEGT

While decrementing signed scalar greater than scalar

Generate a predicate that starting from the highest numbered element is true while the decrementing value of the first, signed scalar operand is greater than the second scalar operand and false thereafter down to the lowest numbered element.
The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.
The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.



WHILEGT <Pd>.<T>, <R><n>, <R><m>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = FALSE;
SVECmp op = Cmp_GT;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<R> Is a width specifier, encoded in “sf”:

sf	<R>
0	W
1	X

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;

for e = elements-1 downto 0
    boolean cond;
    case op of
        when Cmp_GT cond = (Int(operand1, unsigned) >  Int(operand2, unsigned));
        when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));

    last = last && cond;
    ElemP[result, e, esize] = if last then '1' else '0';
    operand1 = operand1 - 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

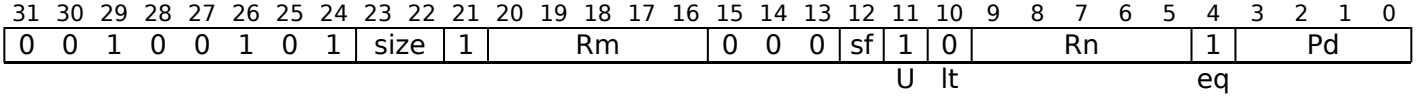
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

WHILEHI

While decrementing unsigned scalar higher than scalar

Generate a predicate that starting from the highest numbered element is true while the decrementing value of the first, unsigned scalar operand is higher than the second scalar operand and false thereafter down to the lowest numbered element.
The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.
The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.



WHILEHI <Pd>.<T>, <R><n>, <R><m>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = TRUE;
SVECmp op = Cmp_GT;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<R> Is a width specifier, encoded in “sf”:

sf	<R>
0	W
1	X

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;

for e = elements-1 downto 0
    boolean cond;
    case op of
        when Cmp_GT cond = (Int(operand1, unsigned) >  Int(operand2, unsigned));
        when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));

    last = last && cond;
    ElemP[result, e, esize] = if last then '1' else '0';
    operand1 = operand1 - 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

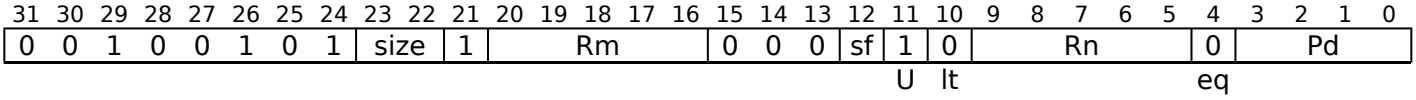
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

WHILEHS

While decrementing unsigned scalar higher or same as scalar

Generate a predicate that starting from the highest numbered element is true while the decrementing value of the first, unsigned scalar operand is higher or same as the second scalar operand and false thereafter down to the lowest numbered element.
If the second scalar operand is equal to the minimum unsigned integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.
The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is decremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.
The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.



WHILEHS <Pd>.<T>, <R><n>, <R><m>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = TRUE;
SVECmp op = Cmp_GE;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<R> Is a width specifier, encoded in “sf”:

sf	<R>
0	W
1	X

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;

for e = elements-1 downto 0
    boolean cond;
    case op of
        when Cmp_GT cond = (Int(operand1, unsigned) >  Int(operand2, unsigned));
        when Cmp_GE cond = (Int(operand1, unsigned) >= Int(operand2, unsigned));

    last = last && cond;
    ElemP[result, e, esize] = if last then '1' else '0';
    operand1 = operand1 - 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

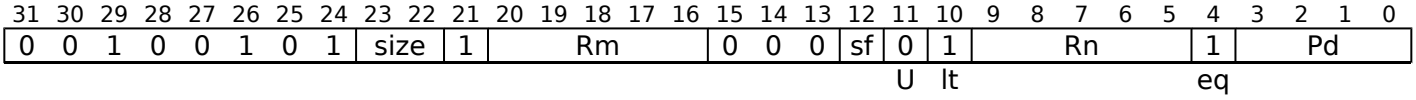
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

WHILELE

While incrementing signed scalar less than or equal to scalar

Generate a predicate that starting from the lowest numbered element is true while the incrementing value of the first, signed scalar operand is less than or equal to the second scalar operand and false thereafter up to the highest numbered element.
If the second scalar operand is equal to the maximum signed integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.
The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.
The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.



WHILELE <Pd>.<T>, <R><n>, <R><m>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = FALSE;
SVECmp op = Cmp_LE;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<R> Is a width specifier, encoded in “sf”:

sf	<R>
0	W
1	X

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;

for e = 0 to elements-1
    boolean cond;
    case op of
        when Cmp_LT cond = (Int(operand1, unsigned) <  Int(operand2, unsigned));
        when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));

    last = last && cond;
    ElemP[result, e, esize] = if last then '1' else '0';
    operand1 = operand1 + 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

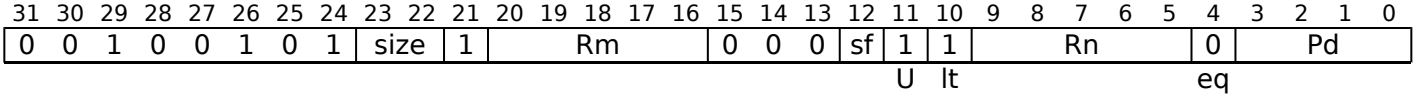
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

WHILELO

While incrementing unsigned scalar lower than scalar

Generate a predicate that starting from the lowest numbered element is true while the incrementing value of the first, unsigned scalar operand is lower than the second scalar operand and false thereafter up to the highest numbered element.
The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.
The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.



WHILELO <Pd>.<T>, <R><n>, <R><m>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = TRUE;
SVECmp op = Cmp_LT;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<R> Is a width specifier, encoded in “sf”:

sf	<R>
0	W
1	X

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;

for e = 0 to elements-1
    boolean cond;
    case op of
        when Cmp_LT cond = (Int(operand1, unsigned) <  Int(operand2, unsigned));
        when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));

    last = last && cond;
    ElemP[result, e, esize] = if last then '1' else '0';
    operand1 = operand1 + 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

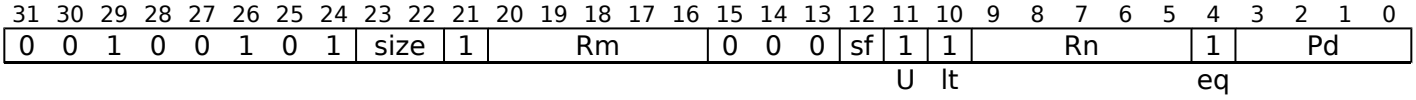
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

WHILELS

While incrementing unsigned scalar lower or same as scalar

Generate a predicate that starting from the lowest numbered element is true while the incrementing value of the first, unsigned scalar operand is lower or same as the second scalar operand and false thereafter up to the highest numbered element.
If the second scalar operand is equal to the maximum unsigned integer value then a condition which includes an equality test can never fail and the result will be an all-true predicate.
The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.
The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.



WHILELS <Pd>.<T>, <R><n>, <R><m>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = TRUE;
SVECmp op = Cmp_LE;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<R> Is a width specifier, encoded in “sf”:

sf	<R>
0	W
1	X

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;

for e = 0 to elements-1
    boolean cond;
    case op of
        when Cmp_LT cond = (Int(operand1, unsigned) <  Int(operand2, unsigned));
        when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));

    last = last && cond;
    ElemP[result, e, esize] = if last then '1' else '0';
    operand1 = operand1 + 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

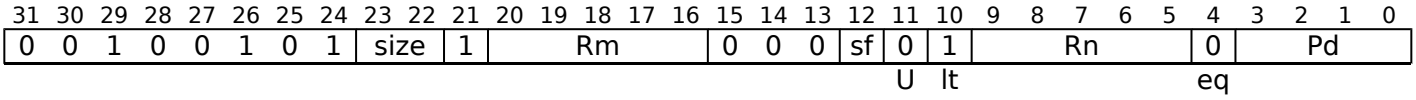
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

WHILELT

While incrementing signed scalar less than scalar

Generate a predicate that starting from the lowest numbered element is true while the incrementing value of the first, signed scalar operand is less than the second scalar operand and false thereafter up to the highest numbered element. The full width of the scalar operands is significant for the purposes of comparison, and the full width first operand is incremented by one for each destination predicate element, irrespective of the predicate result element size. The first general-purpose source register is not itself updated.

The predicate result is placed in the predicate destination register. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.



WHILELT <Pd>.<T>, <R><n>, <R><m>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer rsize = 32 << UInt(sf);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
boolean unsigned = FALSE;
SVECmp op = Cmp_LT;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<R> Is a width specifier, encoded in “sf”:

sf	<R>
0	W
1	X

<n> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rn" field.

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(rsize) operand1 = X[n, rsize];
bits(rsize) operand2 = X[m, rsize];
bits(PL) result;
boolean last = TRUE;

for e = 0 to elements-1
    boolean cond;
    case op of
        when Cmp_LT cond = (Int(operand1, unsigned) <  Int(operand2, unsigned));
        when Cmp_LE cond = (Int(operand1, unsigned) <= Int(operand2, unsigned));

    last = last && cond;
    ElemP[result, e, esize] = if last then '1' else '0';
    operand1 = operand1 + 1;

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

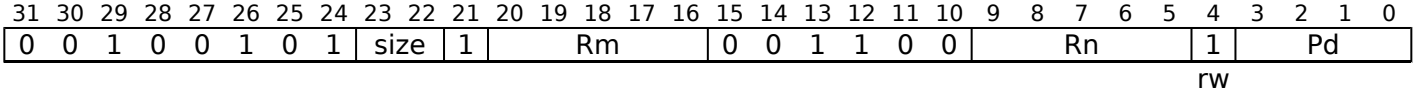
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

WHILERW

While free of read-after-write conflicts

This instruction checks two addresses for a conflict or overlap between address ranges of the form [addr,addr+VL÷8), where VL is the accessible vector length in bits, that could result in a loop-carried dependency through memory due to the use of these addresses by contiguous load and store instructions within the same iteration of a loop. Generate a predicate whose elements are true while the addresses cannot conflict within the same iteration, and false thereafter. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.



WHILERW <Pd>.<T>, <Xn>, <Xm>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Xn> Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(64) src1 = X[n, 64];
bits(64) src2 = X[m, 64];
integer operand1 = UInt(src1);
integer operand2 = UInt(src2);
bits(PL) result;

integer diff = Abs(operand2 - operand1) DIV (esize DIV 8);
for e = 0 to elements-1
    if diff == 0 || e < diff then
        ElemP[result, e, esize] = '1';
    else
        ElemP[result, e, esize] = '0';

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

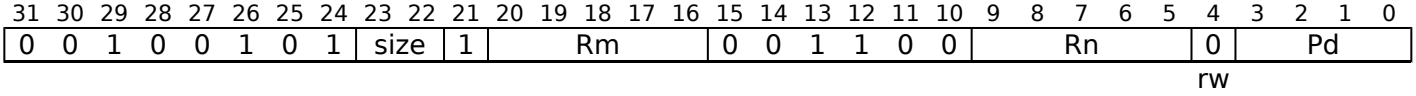
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

WHILEWR

While free of write-after-read/write conflicts

This instruction checks two addresses for a conflict or overlap between address ranges of the form [addr,addr+VL÷8), where VL is the accessible vector length in bits, that could result in a loop-carried dependency through memory due to the use of these addresses by contiguous load and store instructions within the same iteration of a loop. Generate a predicate whose elements are true while the addresses cannot conflict within the same iteration, and false thereafter. Sets the FIRST (N), NONE (Z), !LAST (C) condition flags based on the predicate result, and the V flag to zero.



WHILEWR <Pd>.<T>, <Xn>, <Xm>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer d = UInt(Pd);
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Xn> Is the 64-bit name of the first source general-purpose register, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the second source general-purpose register, encoded in the "Rm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer elements = VL DIV esize;
bits(PL) mask = Ones(PL);
bits(64) src1 = X[n, 64];
bits(64) src2 = X[m, 64];
integer operand1 = UInt(src1);
integer operand2 = UInt(src2);
bits(PL) result;

integer diff = (operand2 - operand1) DIV (esize DIV 8);
for e = 0 to elements-1
    if diff <= 0 || e < diff then
        ElemP[result, e, esize] = '1';
    else
        ElemP[result, e, esize] = '0';

PSTATE.<N,Z,C,V> = PredTest(mask, result, esize);
P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

WRFFR

Write the first-fault register

Read the source predicate register and place in the first-fault register (FFR). This instruction is intended to restore a saved FFR and is not recommended for general use by applications.

This instruction requires that the source predicate contains a MONOTONIC predicate value, in which starting from bit 0 there are zero or more 1 bits, followed only by 0 bits in any remaining bit positions. If the source is not a monotonic predicate value, then the resulting value in the FFR will be UNPREDICTABLE. It is not possible to generate a non-monotonic value in FFR when using SETFFR followed by first-fault or non-fault loads.

This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled at the current Exception level.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	0	0	1	0	1	0	0	0	1	0	0	1	0	0	0	Pn				0	0	0	0	0

WRFFR <Pn>.B

```
if !HaveSVE() then UNDEFINED;
integer n = UInt(Pn);
```

Assembler Symbols

<Pn> Is the name of the source scalable predicate register, encoded in the "Pn" field.

Operation

```
CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
bits(PL) operand = P[n, PL];

constant integer hsb = HighestSetBit(operand);
if hsb < 0 || IsOnes(operand<hsb:0>) then
    FFR[PL] = operand;
else // not a monotonic predicate
    FFR[PL] = bits(PL) UNKNOWN;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

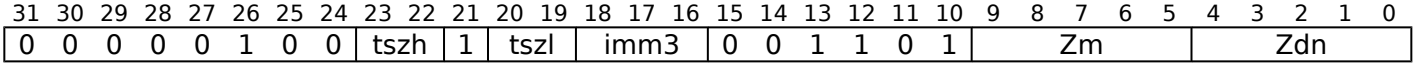
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

XAR

Bitwise exclusive OR and rotate right by immediate

Bitwise exclusive OR the corresponding elements of the first and second source vectors, then rotate each result element right by an immediate amount. The final results are destructively placed in the corresponding elements of the destination and first source vector. This instruction is unpredicated.



XAR <Zdn>.<T>, <Zdn>.<T>, <Zm>.<T>, #<const>

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
bits(4) tsize = tszh:tszl;
integer esize;
case tsize of
    when '0000' UNDEFINED;
    when '0001' esize = 8;
    when '001x' esize = 16;
    when '01xx' esize = 32;
    when '1xxx' esize = 64;
integer m = UInt(Zm);
integer dn = UInt(Zdn);
integer rot = (2 * esize) - UInt(tsize:imm3);
```

Assembler Symbols

<Zdn> Is the name of the first source and destination scalable vector register, encoded in the "Zdn" field.

<T> Is the size specifier, encoded in “tszh:tszl”:

tszh	tszl	<T>
00	00	RESERVED
00	01	B
00	1x	H
01	xx	S
1x	xx	D

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

<const> Is the immediate shift amount, in the range 1 to number of bits per element, encoded in "tsz:imm3".

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer elements = VL DIV esize;
bits(VL) operand1 = Z[dn, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result;

for e = 0 to elements-1
    bits(esize) element1 = Elem[operand1, e, esize];
    bits(esize) element2 = Elem[operand2, e, esize];
    Elem[result, e, esize] = ROR(element1 EOR element2, rot);
Z[dn, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

This instruction might be immediately preceded in program order by a `MOVPRFX` instruction. The `MOVPRFX` instruction must conform to all of the following requirements, otherwise the behavior of the `MOVPRFX` and this instruction is UNPREDICTABLE:

- The `MOVPRFX` instruction must be unpredicated.
- The `MOVPRFX` instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

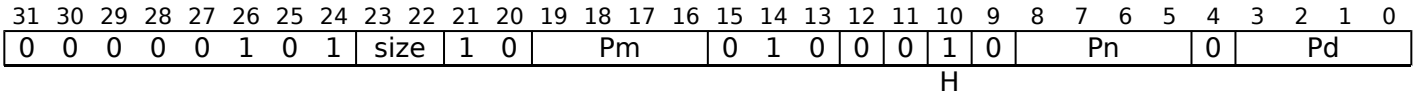
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ZIP1, ZIP2 (predicates)

Interleave elements from two half predicates

Interleave alternating elements from the lowest or highest halves of the first and second source predicates and place in elements of the destination predicate. This instruction is unpredicated.
It has encodings from 2 classes: High halves and Low halves

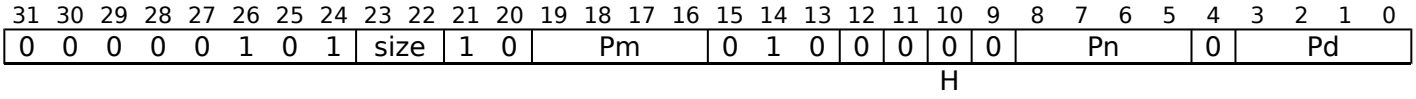
High halves



ZIP2 <Pd>.<T>, <Pn>.<T>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
integer part = 1;
```

Low halves



ZIP1 <Pd>.<T>, <Pn>.<T>, <Pm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Pn);
integer m = UInt(Pm);
integer d = UInt(Pd);
integer part = 0;
```

Assembler Symbols

<Pd> Is the name of the destination scalable predicate register, encoded in the "Pd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Pn> Is the name of the first source scalable predicate register, encoded in the "Pn" field.

<Pm> Is the name of the second source scalable predicate register, encoded in the "Pm" field.

Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (esize * 2);
bits(PL) operand1 = P[n, PL];
bits(PL) operand2 = P[m, PL];
bits(PL) result;

integer base = part * pairs;
for p = 0 to pairs-1
    Elem[result, 2*p+0, esize DIV 8] = Elem[operand1, base+p, esize DIV 8];
    Elem[result, 2*p+1, esize DIV 8] = Elem[operand2, base+p, esize DIV 8];

P[d, PL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

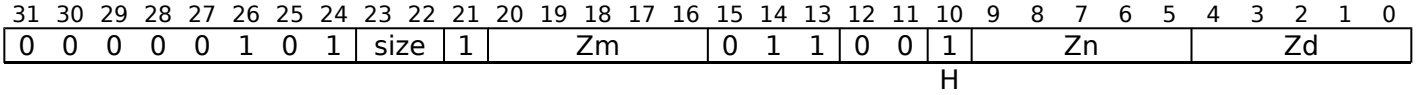
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ZIP1, ZIP2 (vectors)

Interleave elements from two half vectors

Interleave alternating elements from the lowest or highest halves of the first and second source vectors and place in elements of the destination vector. This instruction is unpredicated. The 128-bit element variant of this instruction requires that the current vector length is at least 256 bits, and if the current vector length is not an integer multiple of 256 bits then the trailing bits are set to zero.
ID_AA64ZFR0_EL1.F64MM indicates whether the 128-bit element variant of the instruction is implemented.
It has encodings from 4 classes: High halves , High halves (quadwords) , Low halves and Low halves (quadwords)

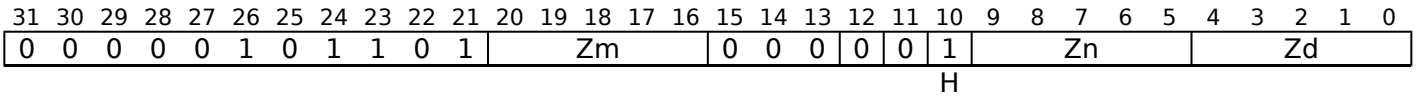
High halves



ZIP2 <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;
```

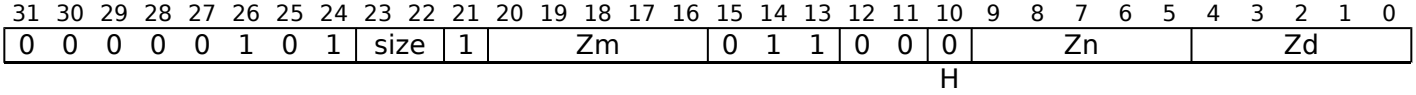
High halves (quadwords)
(FEAT_F64MM)



ZIP2 <Zd>.Q, <Zn>.Q, <Zm>.Q

```
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
constant integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 1;
```

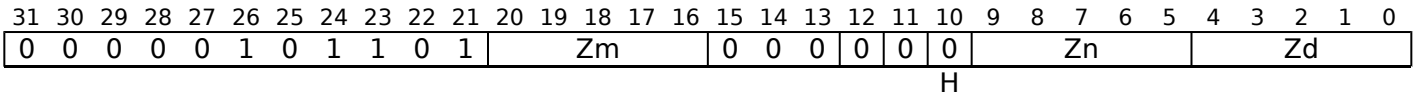
Low halves



ZIP1 <Zd>.<T>, <Zn>.<T>, <Zm>.<T>

```
if !HaveSVE() && !HaveSME() then UNDEFINED;
constant integer esize = 8 << UInt(size);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;
```

Low halves (quadwords)
(FEAT_F64MM)



ZIP1 <Zd>.Q, <Zn>.Q, <Zm>.Q

```
if !HaveSVE() || !HaveSVEFP64MatMulExt() then UNDEFINED;
constant integer esize = 128;
integer n = UInt(Zn);
integer m = UInt(Zm);
integer d = UInt(Zd);
integer part = 0;
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<T> Is the size specifier, encoded in “size”:

size	<T>
00	B
01	H
10	S
11	D

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
if esize < 128 then CheckSVEEnabled(); else CheckNonStreamingSVEEnabled();
constant integer VL = CurrentVL;
if VL < esize * 2 then UNDEFINED;
constant integer pairs = VL DIV (esize * 2);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) result = Zeros(VL);

integer base = part * pairs;
for p = 0 to pairs-1
    Elem[result, 2*p+0, esize] = Elem[operand1, base+p, esize];
    Elem[result, 2*p+1, esize] = Elem[operand2, base+p, esize];

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

A64 -- SME Instructions (alphabetic order)

ADDHA: Add horizontally vector elements to ZA tile.

ADDSPL: Add multiple of Streaming SVE predicate register size to scalar register.

ADDSVL: Add multiple of Streaming SVE vector register size to scalar register.

ADDVA: Add vertically vector elements to ZA tile.

BFMOPA: BFloat16 sum of outer products and accumulate.

BFMOPS: BFloat16 sum of outer products and subtract.

FMOPA (non-widening): Floating-point outer product and accumulate.

FMOPA (widening): Half-precision floating-point sum of outer products and accumulate.

FMOPS (non-widening): Floating-point outer product and subtract.

FMOPS (widening): Half-precision floating-point sum of outer products and subtract.

LD1B: Contiguous load of bytes to 8-bit element ZA tile slice.

LD1D: Contiguous load of doublewords to 64-bit element ZA tile slice.

LD1H: Contiguous load of halfwords to 16-bit element ZA tile slice.

LD1Q: Contiguous load of quadwords to 128-bit element ZA tile slice.

LD1W: Contiguous load of words to 32-bit element ZA tile slice.

LDR: Load vector to ZA array.

MOV (tile to vector): Move ZA tile slice to vector register: an alias of MOVA (tile to vector).

MOV (vector to tile): Move vector register to ZA tile slice: an alias of MOVA (vector to tile).

MOVA (tile to vector): Move ZA tile slice to vector register.

MOVA (vector to tile): Move vector register to ZA tile slice.

RDSVL: Read multiple of Streaming SVE vector register size to scalar register.

SMOPA: Signed integer sum of outer products and accumulate.

SMOPS: Signed integer sum of outer products and subtract.

ST1B: Contiguous store of bytes from 8-bit element ZA tile slice.

ST1D: Contiguous store of doublewords from 64-bit element ZA tile slice.

ST1H: Contiguous store of halfwords from 16-bit element ZA tile slice.

ST1Q: Contiguous store of quadwords from 128-bit element ZA tile slice.

ST1W: Contiguous store of words from 32-bit element ZA tile slice.

STR: Store vector from ZA array.

SUMOPA: Signed by unsigned integer sum of outer products and accumulate.

SUMOPS: Signed by unsigned integer sum of outer products and subtract.

UMOPA: Unsigned integer sum of outer products and accumulate.

UMOPS: Unsigned integer sum of outer products and subtract.

USMOPA: Unsigned by signed integer sum of outer products and accumulate.

USMOPS: Unsigned by signed integer sum of outer products and subtract.

ZERO: Zero a list of 64-bit element ZA tiles.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

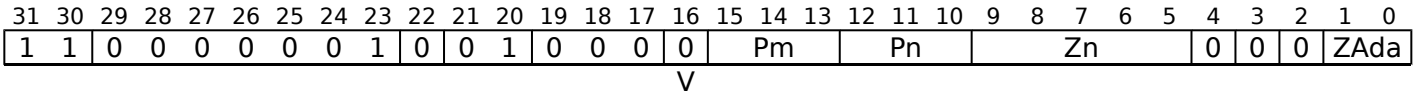
ADDHA

Add horizontally vector elements to ZA tile

Add each element of the source vector to the corresponding active element of each horizontal slice of a ZA tile. The tile elements are predicated by a pair of governing predicates. An element of a horizontal slice is considered active if its corresponding element in the second governing predicate is TRUE and the element corresponding to its horizontal slice number in the first governing predicate is TRUE. Inactive elements in the destination tile remain unmodified.
ID_AA64SMFR0_EL1.I16I64 indicates whether the 64-bit integer variant is implemented.
It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SME)

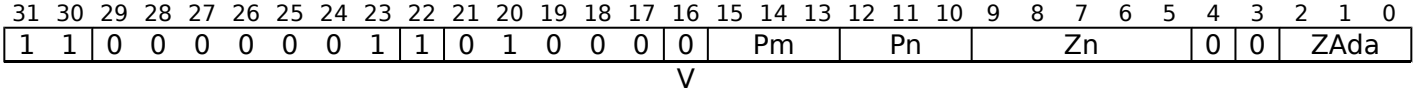


ADDHA <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.S

```
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer da = UInt(ZAda);
```

64-bit

(FEAT_SME_I16I64)



ADDHA <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.D

```
if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer da = UInt(ZAda);
```

Assembler Symbols

<ZAda>	For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field. For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.
<Pn>	Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
<Pm>	Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
<Zn>	Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand_src = Z[n, VL];
bits(dim*dim*esize) operand_acc = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;

for col = 0 to dim-1
    bits(esize) element = Elem[operand_src, col, esize];
    for row = 0 to dim-1
        bits(esize) res = Elem[operand_acc, row*dim+col, esize];
        if ElemP[mask1, row, esize] == '1' && ElemP[mask2, col, esize] == '1' then
            res = res + element;
        Elem[result, row*dim+col, esize] = res;

ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.

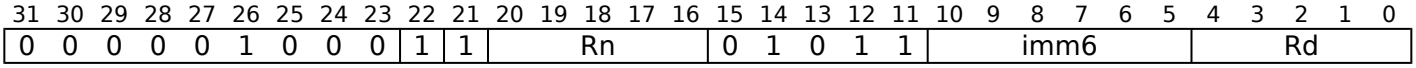
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ADDSPL

Add multiple of Streaming SVE predicate register size to scalar register

Add the Streaming SVE predicate register size in bytes multiplied by an immediate in the range -32 to 31 to the 64-bit source general-purpose register or current stack pointer and place the result in the 64-bit destination general-purpose register or current stack pointer.
This instruction does not require the PE to be in Streaming SVE mode.

SME
(FEAT_SME)



ADDSPL <Xd|SP>, <Xn|SP>, #<imm>

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer d = UInt(Rd);
integer imm = SInt(imm6);
```

Assembler Symbols

<Xd|SP>	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xn|SP>	Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<imm>	Is the signed immediate operand, in the range -32 to 31, encoded in the "imm6" field.

Operation

```
CheckSMEEnabled();
constant integer svl = SVL;
integer len = imm * (svl DIV 64);
bits(64) operand1 = if n == 31 then SP[] else X[n, 64];
bits(64) result = operand1 + len;

if d == 31 then
    SP[] = result;
else
    X[d, 64] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

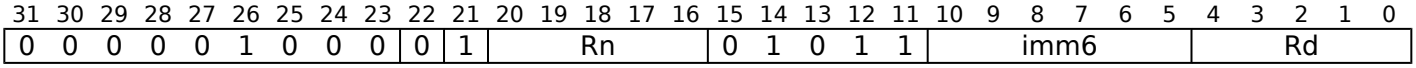
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ADDSVL

Add multiple of Streaming SVE vector register size to scalar register

Add the Streaming SVE vector register size in bytes multiplied by an immediate in the range -32 to 31 to the 64-bit source general-purpose register or current stack pointer, and place the result in the 64-bit destination general-purpose register or current stack pointer.
This instruction does not require the PE to be in Streaming SVE mode.

SME
(FEAT_SME)



ADDSVL <Xd|SP>, <Xn|SP>, #<imm>

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer d = UInt(Rd);
integer imm = SInt(imm6);
```

Assembler Symbols

<Xd|SP>	Is the 64-bit name of the destination general-purpose register or stack pointer, encoded in the "Rd" field.
<Xn|SP>	Is the 64-bit name of the source general-purpose register or stack pointer, encoded in the "Rn" field.
<imm>	Is the signed immediate operand, in the range -32 to 31, encoded in the "imm6" field.

Operation

```
CheckSMEEnabled();
constant integer svl = SVL;
integer len = imm * (svl DIV 8);
bits(64) operand1 = if n == 31 then SP[] else X[n, 64];
bits(64) result = operand1 + len;

if d == 31 then
    SP[] = result;
else
    X[d, 64] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

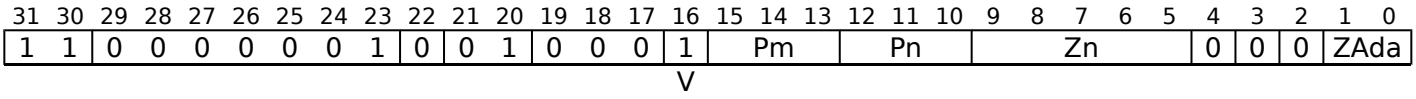
ADDVA

Add vertically vector elements to ZA tile

Add each element of the source vector to the corresponding active element of each vertical slice of a ZA tile. The tile elements are predicated by a pair of governing predicates. An element of a vertical slice is considered active if its corresponding element in the first governing predicate is TRUE and the element corresponding to its vertical slice number in the second governing predicate is TRUE. Inactive elements in the destination tile remain unmodified.
ID_AA64SMFR0_EL1.I16I64 indicates whether the 64-bit integer variant is implemented.
It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SME)

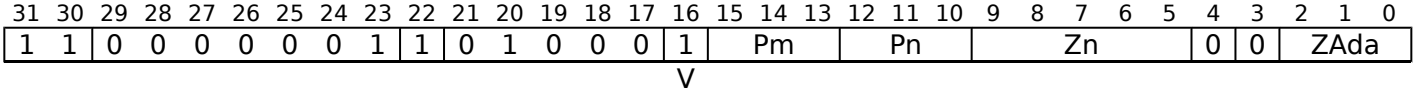


ADDVA <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.S

```
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer da = UInt(ZAda);
```

64-bit

(FEAT_SME_I16I64)



ADDVA <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.D

```
if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer da = UInt(ZAda);
```

Assembler Symbols

<ZAda>	For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field. For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.
<Pn>	Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
<Pm>	Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
<Zn>	Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand_src = Z[n, VL];
bits(dim*dim*esize) operand_acc = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;

for row = 0 to dim-1
    bits(esize) element = Elem[operand_src, row, esize];
    for col = 0 to dim-1
        bits(esize) res = Elem[operand_acc, row*dim+col, esize];
        if ElemP[mask1, row, esize] == '1' && ElemP[mask2, col, esize] == '1' then
            res = res + element;
        Elem[result, row*dim+col, esize] = res;

ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.

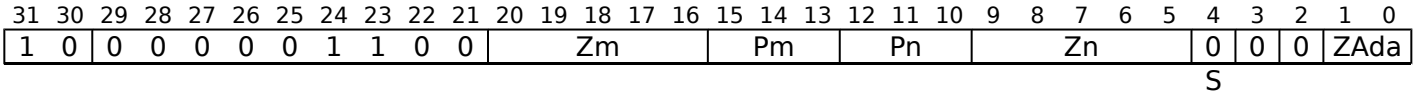
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BFMOPA

BFloat16 sum of outer products and accumulate

The BFloat16 floating-point sum of outer products and accumulate instruction works with a 32-bit element ZA tile.
This instruction multiplies the $SVL_S \times 2$ sub-matrix of BFloat16 values held in the first source vector by the $2 \times SVL_S$ sub-matrix of BFloat16 values in the second source vector.
Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is Inactive it is treated as having the value +0.0, but if both pairs of source vector elements that correspond to a 32-bit destination element contain Inactive elements, then the destination element remains unmodified.
The resulting $SVL_S \times SVL_S$ single-precision floating-point sum of outer products is then destructively added to the single-precision floating-point destination tile. This is equivalent to performing a 2-way dot product and accumulate to each of the destination tile elements.
Each 32-bit container of the first source vector holds 2 consecutive column elements of each row of a $SVL_S \times 2$ sub-matrix. Similarly, each 32-bit container of the second source vector holds 2 consecutive row elements of each column of a $2 \times SVL_S$ sub-matrix.
This instruction follows SME BFloat16 numerical behaviors.

SME
(FEAT_SME)



BFMOPA <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

```
if !HaveSME() then UNDEFINED;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
```

Assembler Symbols

<ZAda>	Is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
<Pn>	Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
<Pm>	Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV 32;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*32) operand3 = ZAtile[da, 32, dim*dim*32];
bits(dim*dim*32) result;

for row = 0 to dim-1
    for col = 0 to dim-1
        // determine row/col predicates
        boolean prow_0 = (ElemP[mask1, 2*row + 0, 16] == '1');
        boolean prow_1 = (ElemP[mask1, 2*row + 1, 16] == '1');
        boolean pcol_0 = (ElemP[mask2, 2*col + 0, 16] == '1');
        boolean pcol_1 = (ElemP[mask2, 2*col + 1, 16] == '1');

        bits(32) sum = Elem[operand3, row*dim+col, 32];
        if (prow_0 && pcol_0) || (prow_1 && pcol_1) then
            bits(16) erow_0 = (if prow_0 then Elem[operand1, 2*row + 0, 16] else FPZero('0', 16));
            bits(16) erow_1 = (if prow_1 then Elem[operand1, 2*row + 1, 16] else FPZero('0', 16));
            bits(16) ecol_0 = (if pcol_0 then Elem[operand2, 2*col + 0, 16] else FPZero('0', 16));
            bits(16) ecol_1 = (if pcol_1 then Elem[operand2, 2*col + 1, 16] else FPZero('0', 16));
            if sub_op then
                if prow_0 then erow_0 = BFNeg(erow_0);
                if prow_1 then erow_1 = BFNeg(erow_1);
            sum = BFDotAdd(sum, erow_0, erow_1, ecol_0, ecol_1, FPCR[]);

        Elem[result, row*dim+col, 32] = sum;

ZAtile[da, 32, dim*dim*32] = result;
```

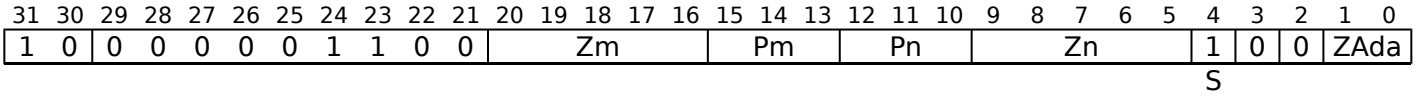
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

BFMOPS

BFloat16 sum of outer products and subtract

The BFloat16 floating-point sum of outer products and subtract instruction works with a 32-bit element ZA tile.
This instruction multiplies the $SVL_S \times 2$ sub-matrix of BFloat16 values held in the first source vector by the $2 \times SVL_S$ sub-matrix of BFloat16 values in the second source vector.
Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is Inactive it is treated as having the value +0.0, but if both pairs of source vector elements that correspond to a 32-bit destination element contain Inactive elements, then the destination element remains unmodified.
The resulting $SVL_S \times SVL_S$ single-precision floating-point sum of outer products is then destructively subtracted from the single-precision floating-point destination tile. This is equivalent to performing a 2-way dot product and subtract from each of the destination tile elements.
Each 32-bit container of the first source vector holds 2 consecutive column elements of each row of a $SVL_S \times 2$ sub-matrix. Similarly, each 32-bit container of the second source vector holds 2 consecutive row elements of each column of a $2 \times SVL_S$ sub-matrix.
This instruction follows SME BFloat16 numerical behaviors.

SME
(FEAT_SME)



BFMOPS <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

```
if !HaveSME() then UNDEFINED;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
```

Assembler Symbols

<ZAda>	Is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
<Pn>	Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
<Pm>	Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV 32;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*32) operand3 = ZAtile[da, 32, dim*dim*32];
bits(dim*dim*32) result;

for row = 0 to dim-1
    for col = 0 to dim-1
        // determine row/col predicates
        boolean prow_0 = (ElemP[mask1, 2*row + 0, 16] == '1');
        boolean prow_1 = (ElemP[mask1, 2*row + 1, 16] == '1');
        boolean pcol_0 = (ElemP[mask2, 2*col + 0, 16] == '1');
        boolean pcol_1 = (ElemP[mask2, 2*col + 1, 16] == '1');

        bits(32) sum = Elem[operand3, row*dim+col, 32];
        if (prow_0 && pcol_0) || (prow_1 && pcol_1) then
            bits(16) erow_0 = (if prow_0 then Elem[operand1, 2*row + 0, 16] else FPZero('0', 16));
            bits(16) erow_1 = (if prow_1 then Elem[operand1, 2*row + 1, 16] else FPZero('0', 16));
            bits(16) ecol_0 = (if pcol_0 then Elem[operand2, 2*col + 0, 16] else FPZero('0', 16));
            bits(16) ecol_1 = (if pcol_1 then Elem[operand2, 2*col + 1, 16] else FPZero('0', 16));
            if sub_op then
                if prow_0 then erow_0 = BFNeg(erow_0);
                if prow_1 then erow_1 = BFNeg(erow_1);
            sum = BFDotAdd(sum, erow_0, erow_1, ecol_0, ecol_1, FPCR[]);

        Elem[result, row*dim+col, 32] = sum;

ZAtile[da, 32, dim*dim*32] = result;
```

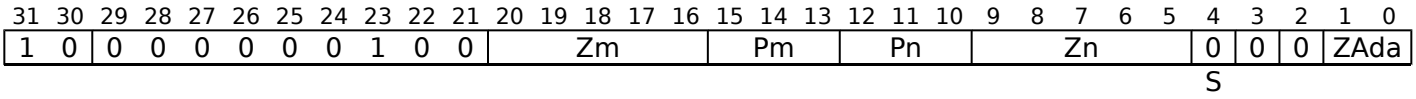
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMOPA (non-widening)

Floating-point outer product and accumulate

The single-precision variant works with a 32-bit element ZA tile.
The double-precision variant works with a 64-bit element ZA tile.
These instructions generate an outer product of the first source vector and the second source vector. In case of the single-precision variant, the first source is $SVL_S \times 1$ vector and the second source is $1 \times SVL_S$ vector. In case of the double-precision variant, the first source is $SVL_D \times 1$ vector and the second source is $1 \times SVL_D$ vector.
Each source vector is independently predicated by a corresponding governing predicate. When either source vector element is Inactive the corresponding destination tile element remains unmodified.
The resulting outer product, $SVL_S \times SVL_S$ in case of single-precision variant or $SVL_D \times SVL_D$ in case of double-precision variant, is then destructively added to the destination tile. This is equivalent to performing a single multiply-accumulate to each of the destination tile elements.
This instruction follows SME ZA-targeting floating-point behaviors.
ID_AA64SMFR0_EL1.F64F64 indicates whether the double-precision variant is implemented.
It has encodings from 2 classes: Single-precision and Double-precision

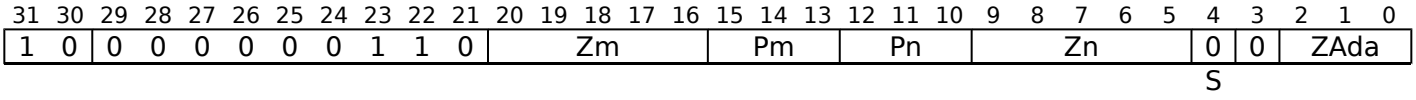
Single-precision
(FEAT_SME)



FMOPA <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.S, <Zm>.S

```
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
```

Double-precision
(FEAT_SME_F64F64)



FMOPA <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.D, <Zm>.D

```
if !HaveSMEF64F64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
```

Assembler Symbols

<ZAda>	For the single-precision variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field. For the double-precision variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.
<Pn>	Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;

for row = 0 to dim-1
    for col = 0 to dim-1
        bits(esize) element1 = Elem[operand1, row, esize];
        bits(esize) element2 = Elem[operand2, col, esize];
        bits(esize) element3 = Elem[operand3, row*dim+col, esize];

        if ElemP[mask1, row, esize] == '1' && ElemP[mask2, col, esize] == '1' then
            if sub_op then element1 = FPNeg(element1);
            Elem[result, row*dim+col, esize] = FPMulAdd_ZA(element3, element1, element2, FPCR[]);
        else
            Elem[result, row*dim+col, esize] = element3;

ZAtile[da, esize, dim*dim*esize] = result;
```

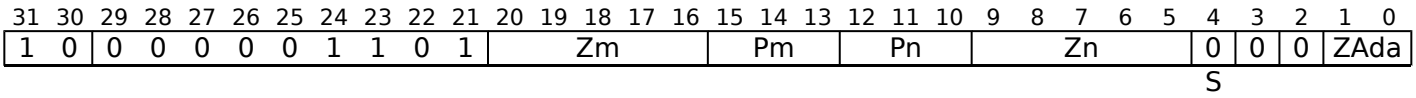
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMOPA (widening)

Half-precision floating-point sum of outer products and accumulate

The half-precision floating-point sum of outer products and accumulate instruction works with a 32-bit element ZA tile.
This instruction widens the $SVL_S \times 2$ sub-matrix of half-precision floating-point values held in the first source vector to single-precision floating-point values and multiplies it by the widened $2 \times SVL_S$ sub-matrix of half-precision floating-point values in the second source vector to single-precision floating-point values.
Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is Inactive it is treated as having the value +0.0, but if both pairs of source vector elements that correspond to a 32-bit destination element contain Inactive elements, then the destination element remains unmodified.
The resulting $SVL_S \times SVL_S$ single-precision floating-point sum of outer products is then destructively added to the single-precision floating-point destination tile. This is equivalent to performing a 2-way dot product and accumulate to each of the destination tile elements.
Each 32-bit container of the first source vector holds 2 consecutive column elements of each row of a $SVL_S \times 2$ sub-matrix. Similarly, each 32-bit container of the second source vector holds 2 consecutive row elements of each column of a $2 \times SVL_S$ sub-matrix.
This instruction follows SME ZA-targeting floating-point behaviors.

SME
(FEAT_SME)



FMOPA <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

```
if !HaveSME() then UNDEFINED;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
```

Assembler Symbols

<ZAda>	Is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
<Pn>	Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
<Pm>	Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV 32;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*32) operand3 = ZAtile[da, 32, dim*dim*32];
bits(dim*dim*32) result;

for row = 0 to dim-1
    for col = 0 to dim-1
        // determine row/col predicates
        boolean prow_0 = (ElemP[mask1, 2*row + 0, 16] == '1');
        boolean prow_1 = (ElemP[mask1, 2*row + 1, 16] == '1');
        boolean pcol_0 = (ElemP[mask2, 2*col + 0, 16] == '1');
        boolean pcol_1 = (ElemP[mask2, 2*col + 1, 16] == '1');

        bits(32) sum = Elem[operand3, row*dim+col, 32];
        if (prow_0 && pcol_0) || (prow_1 && pcol_1) then
            bits(16) erow_0 = (if prow_0 then Elem[operand1, 2*row + 0, 16] else FPZero('0', 16));
            bits(16) erow_1 = (if prow_1 then Elem[operand1, 2*row + 1, 16] else FPZero('0', 16));
            bits(16) ecol_0 = (if pcol_0 then Elem[operand2, 2*col + 0, 16] else FPZero('0', 16));
            bits(16) ecol_1 = (if pcol_1 then Elem[operand2, 2*col + 1, 16] else FPZero('0', 16));
            if sub_op then
                if prow_0 then erow_0 = FPNeg(erow_0);
                if prow_1 then erow_1 = FPNeg(erow_1);
            sum = FPDotAdd_ZA(sum, erow_0, erow_1, ecol_0, ecol_1, FPCR[]);

        Elem[result, row*dim+col, 32] = sum;

ZAtile[da, 32, dim*dim*32] = result;
```

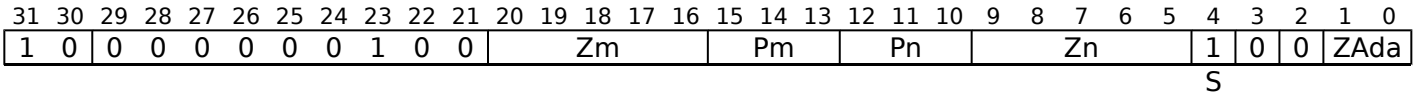
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMOPS (non-widening)

Floating-point outer product and subtract

The single-precision variant works with a 32-bit element ZA tile.
The double-precision variant works with a 64-bit element ZA tile.
These instructions generate an outer product of the first source vector and the second source vector. In case of the single-precision variant, the first source is $SVL_S \times 1$ vector and the second source is $1 \times SVL_S$ vector. In case of the double-precision variant, the first source is $SVL_D \times 1$ vector and the second source is $1 \times SVL_D$ vector.
Each source vector is independently predicated by a corresponding governing predicate. When either source vector element is Inactive the corresponding destination tile element remains unmodified.
The resulting outer product, $SVL_S \times SVL_S$ in case of single-precision variant or $SVL_D \times SVL_D$ in case of double-precision variant, is then destructively subtracted from the destination tile. This is equivalent to performing a single multiply-subtract from each of the destination tile elements.
This instruction follows SME ZA-targeting floating-point behaviors.
ID_AA64SMFR0_EL1.F64F64 indicates whether the double-precision variant is implemented.
It has encodings from 2 classes: Single-precision and Double-precision

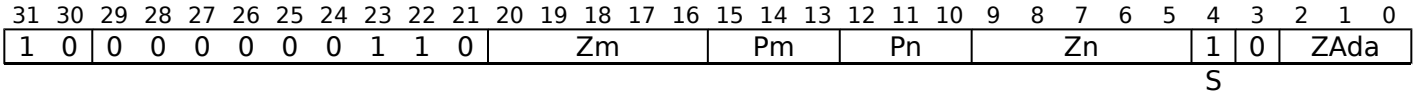
Single-precision
(FEAT_SME)



FMOPS <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.S, <Zm>.S

```
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
```

Double-precision
(FEAT_SME_F64F64)



FMOPS <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.D, <Zm>.D

```
if !HaveSMEF64F64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
```

Assembler Symbols

<ZAda>	For the single-precision variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field. For the double-precision variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.
<Pn>	Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.

<Pm> Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.

<Zn> Is the name of the first source scalable vector register, encoded in the "Zn" field.

<Zm> Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;

for row = 0 to dim-1
    for col = 0 to dim-1
        bits(esize) element1 = Elem[operand1, row, esize];
        bits(esize) element2 = Elem[operand2, col, esize];
        bits(esize) element3 = Elem[operand3, row*dim+col, esize];

        if ElemP[mask1, row, esize] == '1' && ElemP[mask2, col, esize] == '1' then
            if sub_op then element1 = FPNeg(element1);
            Elem[result, row*dim+col, esize] = FPMulAdd_ZA(element3, element1, element2, FPCR[]);
        else
            Elem[result, row*dim+col, esize] = element3;

ZAtile[da, esize, dim*dim*esize] = result;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

FMOPS (widening)

Half-precision floating-point sum of outer products and subtract

The half-precision floating-point sum of outer products and subtract instruction works with a 32-bit element ZA tile.

This instruction widens the $SVL_S \times 2$ sub-matrix of half-precision floating-point values held in the first source vector to single-precision floating-point values and multiplies it by the widened $2 \times SVL_S$ sub-matrix of half-precision floating-point values in the second source vector to single-precision floating-point values.

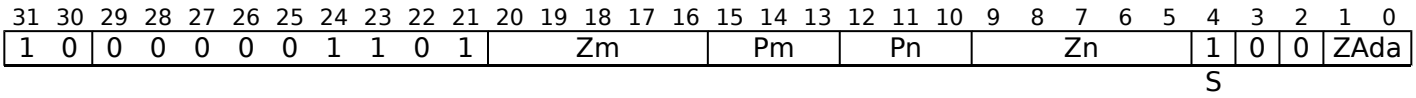
Each source vector is independently predicated by a corresponding governing predicate. When a 16-bit source element is Inactive it is treated as having the value +0.0, but if both pairs of source vector elements that correspond to a 32-bit destination element contain Inactive elements, then the destination element remains unmodified.

The resulting $SVL_S \times SVL_S$ single-precision floating-point sum of outer products is then destructively subtracted from the single-precision floating-point destination tile. This is equivalent to performing a 2-way dot product and subtract from each of the destination tile elements.

Each 32-bit container of the first source vector holds 2 consecutive column elements of each row of a $SVL_S \times 2$ sub-matrix. Similarly, each 32-bit container of the second source vector holds 2 consecutive row elements of each column of a $2 \times SVL_S$ sub-matrix.

This instruction follows SME ZA-targeting floating-point behaviors.

SME
(FEAT_SME)



FMOPS <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

```
if !HaveSME() then UNDEFINED;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
```

Assembler Symbols

<ZAda>	Is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field.
<Pn>	Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
<Pm>	Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV 32;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*32) operand3 = ZAtile[da, 32, dim*dim*32];
bits(dim*dim*32) result;

for row = 0 to dim-1
    for col = 0 to dim-1
        // determine row/col predicates
        boolean prow_0 = (ElemP[mask1, 2*row + 0, 16] == '1');
        boolean prow_1 = (ElemP[mask1, 2*row + 1, 16] == '1');
        boolean pcol_0 = (ElemP[mask2, 2*col + 0, 16] == '1');
        boolean pcol_1 = (ElemP[mask2, 2*col + 1, 16] == '1');

        bits(32) sum = Elem[operand3, row*dim+col, 32];
        if (prow_0 && pcol_0) || (prow_1 && pcol_1) then
            bits(16) erow_0 = (if prow_0 then Elem[operand1, 2*row + 0, 16] else FPZero('0', 16));
            bits(16) erow_1 = (if prow_1 then Elem[operand1, 2*row + 1, 16] else FPZero('0', 16));
            bits(16) ecol_0 = (if pcol_0 then Elem[operand2, 2*col + 0, 16] else FPZero('0', 16));
            bits(16) ecol_1 = (if pcol_1 then Elem[operand2, 2*col + 1, 16] else FPZero('0', 16));
            if sub_op then
                if prow_0 then erow_0 = FPNeg(erow_0);
                if prow_1 then erow_1 = FPNeg(erow_1);
            sum = FPDotAdd_ZA(sum, erow_0, erow_1, ecol_0, ecol_1, FPCR[]);

        Elem[result, row*dim+col, 32] = sum;

ZAtile[da, 32, dim*dim*32] = result;
```

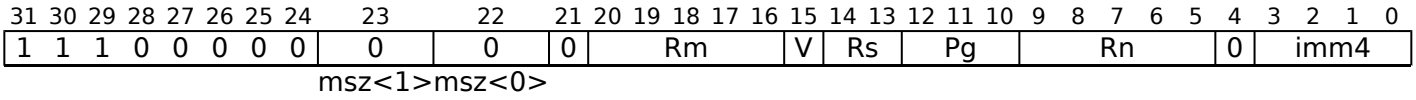
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1B

Contiguous load of bytes to 8-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and an immediate, modulo the number of 8-bit elements in a vector. The immediate is in the range 0 to 15. The memory address is generated by scalar base and optional scalar offset which is added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

SME
(FEAT_SME)



LD1B { ZA0<HV>.B[<Ws>, <imm>] }, <Pg>/Z, [<Xn|SP>{, <Xm>}]

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = 0;
integer imm = UInt(imm4);
constant integer esize = 8;
boolean vertical = V == '1';
```

Assembler Symbols

<HV> Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<imm> Is the slice index offset, in the range 0 to 15, encoded in the "imm4" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(64) offset;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(32) index = X[s, 32];
integer  slice = (UInt(index) + imm) MOD dim;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
    if AnyActiveElement(mask, esize) ||
          ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];
offset = X[m, 64];

for e = 0 to dim - 1
    addr = base + UInt(offset) * mbytes;
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SME];
    else
        Elem[result, e, esize] = Zeros(esize);
    offset = offset + 1;

ZAslice[t, esize, vertical, slice, VL] = result;
```

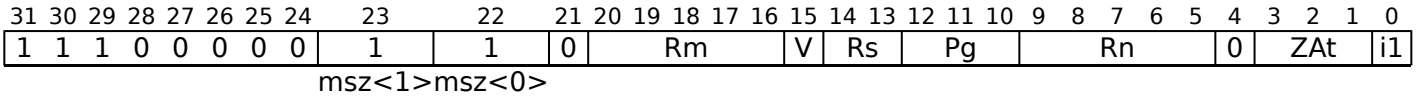
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1D

Contiguous load of doublewords to 64-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and an immediate, modulo the number of 64-bit elements in a vector. The immediate is in the range 0 to 1. The memory address is generated by scalar base and optional scalar offset which is multiplied by 8 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

SME
(FEAT_SME)



LD1D { <ZAt><HV>.D[<Ws>, <imm>] }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #3}]

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = UInt(ZAt);
integer imm = UInt(i1);
constant integer esize = 64;
boolean vertical = V == '1';
```

Assembler Symbols

<ZAt> Is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAt" field.

<HV> Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<imm> Is the slice index offset, in the range 0 to 1, encoded in the "i1" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(64) offset;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(32) index = X[s, 32];
integer  slice = (UInt(index) + imm) MOD dim;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
    if AnyActiveElement(mask, esize) ||
          ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];
offset = X[m, 64];

for e = 0 to dim - 1
    addr = base + UInt(offset) * mbytes;
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SME];
    else
        Elem[result, e, esize] = Zeros(esize);
    offset = offset + 1;

ZAslice[t, esize, vertical, slice, VL] = result;
```

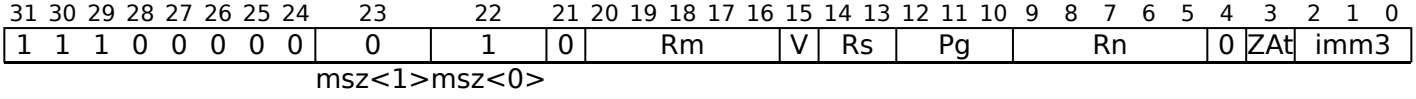
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1H

Contiguous load of halfwords to 16-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and an immediate, modulo the number of 16-bit elements in a vector. The immediate is in the range 0 to 7. The memory address is generated by scalar base and optional scalar offset which is multiplied by 2 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

SME
(FEAT_SME)



LD1H { <ZAt><HV>.H[<Ws>, <imm>] }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #1}]

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = UInt(ZAt);
integer imm = UInt(imm3);
constant integer esize = 16;
boolean vertical = V == '1';
```

Assembler Symbols

<ZAt> Is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAt" field.

<HV> Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<imm> Is the slice index offset, in the range 0 to 7, encoded in the "imm3" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(64) offset;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(32) index = X[s, 32];
integer  slice = (UInt(index) + imm) MOD dim;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
    if AnyActiveElement(mask, esize) ||
          ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];
offset = X[m, 64];

for e = 0 to dim - 1
    addr = base + UInt(offset) * mbytes;
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SME];
    else
        Elem[result, e, esize] = Zeros(esize);
    offset = offset + 1;

ZAslice[t, esize, vertical, slice, VL] = result;
```

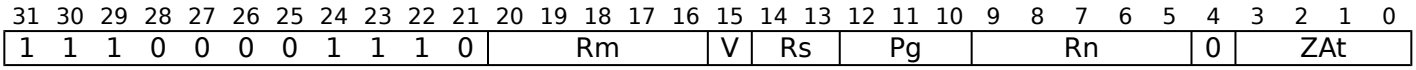
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1Q

Contiguous load of quadwords to 128-bit element ZA tile slice

The slice number in the tile is selected by the slice index register, modulo the number of 128-bit elements in a Streaming SVE vector. The memory address is generated by scalar base and optional scalar offset which is multiplied by 16 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

SME
(FEAT_SME)



LD1Q { <ZAt><HV>.Q[<Ws>, <imm>] }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #4}]

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = UInt(ZAt);
integer imm = 0;
constant integer esize = 128;
boolean vertical = V == '1';
```

Assembler Symbols

<ZAt> Is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the "ZAt" field.

<HV> Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<imm> Is the slice index offset 0.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(64) offset;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(32) index = X[s, 32];
integer  slice = (UInt(index) + imm) MOD dim;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
    if AnyActiveElement(mask, esize) ||
          ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];
offset = X[m, 64];

for e = 0 to dim - 1
    addr = base + UInt(offset) * mbytes;
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SME];
    else
        Elem[result, e, esize] = Zeros(esize);
    offset = offset + 1;

ZAslice[t, esize, vertical, slice, VL] = result;
```

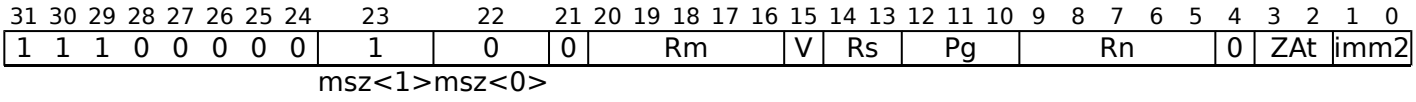
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LD1W

Contiguous load of words to 32-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and an immediate, modulo the number of 32-bit elements in a vector. The immediate is in the range 0 to 3. The memory address is generated by scalar base and optional scalar offset which is multiplied by 4 and added to the base address. Inactive elements will not cause a read from Device memory or signal a fault, and are set to zero in the destination vector.

SME
(FEAT_SME)



LD1W { <ZAt><HV>.S[<Ws>, <imm>] }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #2}]

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = UInt(ZAt);
integer imm = UInt(imm2);
constant integer esize = 32;
boolean vertical = V == '1';
```

Assembler Symbols

<ZAt> Is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAt" field.

<HV> Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<imm> Is the slice index offset, in the range 0 to 3, encoded in the "imm2" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(64) offset;
bits(PL) mask = P[g, PL];
bits(VL) result;
bits(32) index = X[s, 32];
integer  slice = (UInt(index) + imm) MOD dim;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
    if AnyActiveElement(mask, esize) ||
          ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];
offset = X[m, 64];

for e = 0 to dim - 1
    addr = base + UInt(offset) * mbytes;
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = Mem[addr, mbytes, AccType_SME];
    else
        Elem[result, e, esize] = Zeros(esize);
    offset = offset + 1;

ZAslice[t, esize, vertical, slice, VL] = result;
```

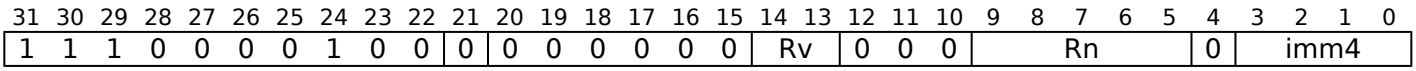
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

LDR

Load vector to ZA array

The ZA array vector is selected by the sum of the vector select register and an immediate, modulo the number of bytes in a Streaming SVE vector. The immediate is in the range 0 to 15. The memory address is generated by scalar base, plus the same optional immediate offset multiplied by the current vector length in bytes. This instruction is unpredicated.
The load is performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then the base register must be aligned to 16 bytes.
This instruction does not require the PE to be in Streaming SVE mode, and it is expected that this instruction will not experience a significant slowdown due to contention with other PEs that are executing in Streaming SVE mode.

SME
(FEAT_SME)



LDR ZA[<Wv>, <imm>], [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer v = UInt('011':Rv);
integer imm = UInt(imm4);
```

Assembler Symbols

<Wv>	Is the 32-bit name of the vector select register W12-W15, encoded in the "Rv" field.
<imm>	Is the vector select offset and optional memory offset, in the range 0 to 15, defaulting to 0, encoded in the "imm4" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckSMEAndZAEnabled();
constant integer svl = SVL;
constant integer dim = svl DIV 8;
bits(64)  base;
integer   offset = imm * dim;
bits(svl) result;
bits(32)  idx = X[v, 32];
integer   vec = (UInt(idx) + imm) MOD dim;

if HaveTME() && TSTATE.depth > 0 then
    FailTransaction(TMFailure_ERR, FALSE);

if n == 31 then
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    CheckSPAlignment();
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n, 64];

boolean aligned = AArch64.CheckAlignment(base + offset, 16, AccType_SME, FALSE);
for e = 0 to dim-1
    Elem[result, e, 8] = AArch64.MemSingle[base + offset, 1, AccType_SME, aligned];
    offset = offset + 1;

ZAvector[vec, svl] = result;
```

MOV (tile to vector)

Move ZA tile slice to vector register

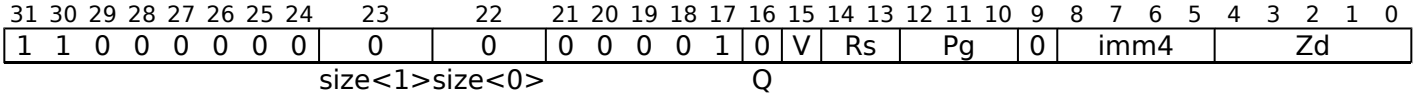
The instruction operates on individual horizontal or vertical slices within a named ZA tile of the specified element size. The slice number within the tile is selected by the sum of the slice index register and an immediate, modulo the number of such elements in a vector. The immediate is in the range 0 to the number of elements in a 128-bit vector segment minus 1.
Inactive elements in the destination vector remain unmodified.

This is an alias of MOVA (tile to vector). This means:

- The encodings in this description are named to match the encodings of MOVA (tile to vector).
- The description of MOVA (tile to vector) gives the operational pseudocode for this instruction.

It has encodings from 5 classes: 8-bit , 16-bit , 32-bit , 64-bit and 128-bit

8-bit



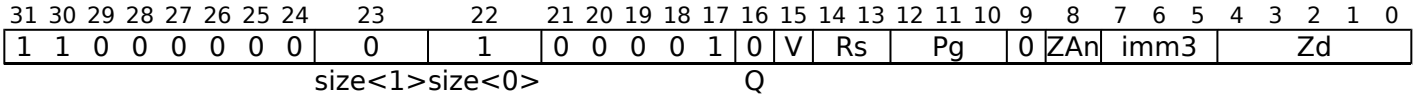
MOV <Zd>.B, <Pg>/M, ZA0<HV>.B[<Ws>, <imm>]

is equivalent to

MOVA <Zd>.B, <Pg>/M, ZA0<HV>.B[<Ws>, <imm>]

and is always the preferred disassembly.

16-bit



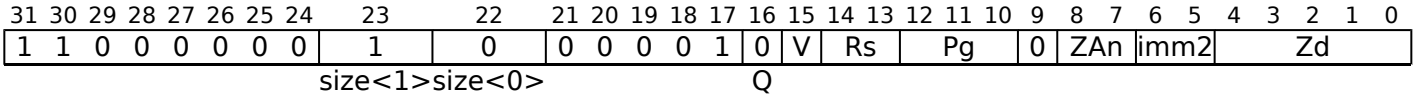
MOV <Zd>.H, <Pg>/M, <ZAn><HV>.H[<Ws>, <imm>]

is equivalent to

MOVA <Zd>.H, <Pg>/M, <ZAn><HV>.H[<Ws>, <imm>]

and is always the preferred disassembly.

32-bit



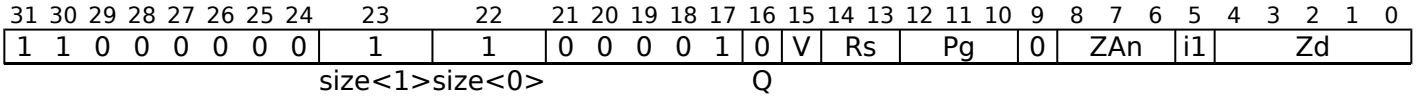
MOV <Zd>.S, <Pg>/M, <ZAn><HV>.S[<Ws>, <imm>]

is equivalent to

MOVA <Zd>.S, <Pg>/M, <ZAn><HV>.S[<Ws>, <imm>]

and is always the preferred disassembly.

64-bit



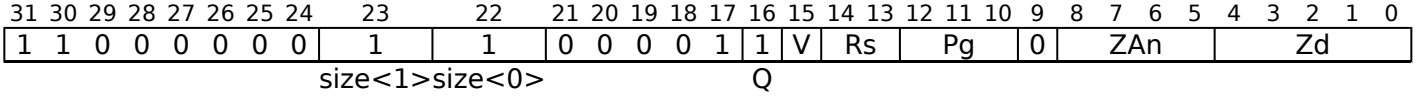
MOV <Zd>.D, <Pg>/M, <ZAn><HV>.D[<Ws>, <imm>]

is equivalent to

MOVA <Zd>.D, <Pg>/M, <ZAn><HV>.D[<Ws>, <imm>]

and is always the preferred disassembly.

128-bit



MOV <Zd>.Q, <Pg>/M, <ZAn><HV>.Q[<Ws>, <imm>]

is equivalent to

MOVA <Zd>.Q, <Pg>/M, <ZAn><HV>.Q[<Ws>, <imm>]

and is always the preferred disassembly.

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<ZAn> For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAn" field.
For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAn" field.
For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAn" field.
For the 128-bit variant: is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the "ZAn" field.

<HV> Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<imm> For the 8-bit variant: is the slice index offset, in the range 0 to 15, encoded in the "imm4" field.
For the 16-bit variant: is the slice index offset, in the range 0 to 7, encoded in the "imm3" field.
For the 32-bit variant: is the slice index offset, in the range 0 to 3, encoded in the "imm2" field.
For the 64-bit variant: is the slice index offset, in the range 0 to 1, encoded in the "i1" field.
For the 128-bit variant: is the slice index offset 0.

Operation

The description of MOVA (tile to vector) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MOV (vector to tile)

Move vector register to ZA tile slice

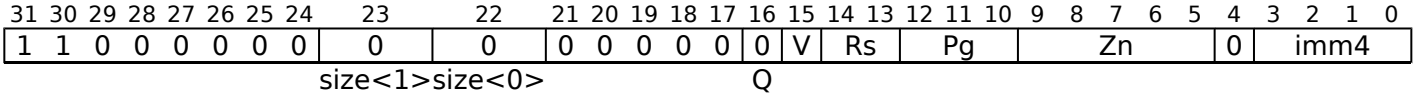
The instruction operates on individual horizontal or vertical slices within a named ZA tile of the specified element size. The slice number within the tile is selected by the sum of the slice index register and an immediate, modulo the number of such elements in a vector. The immediate is in the range 0 to the number of elements in a 128-bit vector segment minus 1.
Inactive elements in the destination slice remain unmodified.

This is an alias of MOVA (vector to tile). This means:

- The encodings in this description are named to match the encodings of MOVA (vector to tile).
- The description of MOVA (vector to tile) gives the operational pseudocode for this instruction.

It has encodings from 5 classes: 8-bit , 16-bit , 32-bit , 64-bit and 128-bit

8-bit



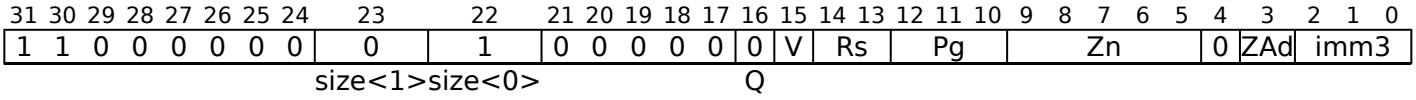
MOV ZA0<HV>.B[<Ws>, <imm>], <Pg>/M, <Zn>.B

is equivalent to

MOVA ZA0<HV>.B[<Ws>, <imm>], <Pg>/M, <Zn>.B

and is always the preferred disassembly.

16-bit



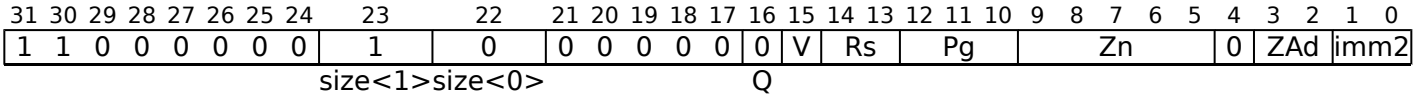
MOV <ZAd><HV>.H[<Ws>, <imm>], <Pg>/M, <Zn>.H

is equivalent to

MOVA <ZAd><HV>.H[<Ws>, <imm>], <Pg>/M, <Zn>.H

and is always the preferred disassembly.

32-bit



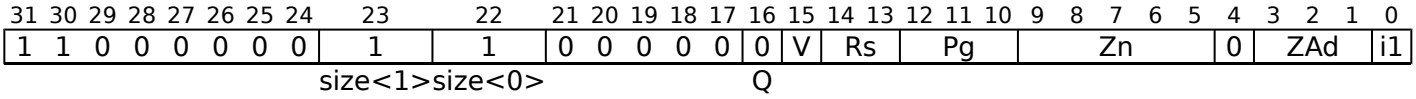
MOV <ZAd><HV>.S[<Ws>, <imm>], <Pg>/M, <Zn>.S

is equivalent to

MOVA <ZAd><HV>.S[<Ws>, <imm>], <Pg>/M, <Zn>.S

and is always the preferred disassembly.

64-bit



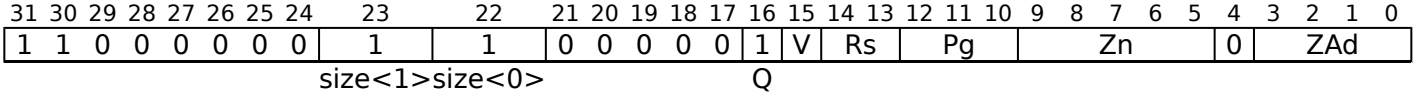
MOV <ZAd><HV>.D[<Ws>, <imm>], <Pg>/M, <Zn>.D

is equivalent to

MOVA <ZAd><HV>.D[<Ws>, <imm>], <Pg>/M, <Zn>.D

and is always the preferred disassembly.

128-bit



MOV <ZAd><HV>.Q[<Ws>, <imm>], <Pg>/M, <Zn>.Q

is equivalent to

MOVA <ZAd><HV>.Q[<Ws>, <imm>], <Pg>/M, <Zn>.Q

and is always the preferred disassembly.

Assembler Symbols

<ZAd> For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAd" field.
For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAd" field.
For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAd" field.
For the 128-bit variant: is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the "ZAd" field.

<HV> Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<imm> For the 8-bit variant: is the slice index offset, in the range 0 to 15, encoded in the "imm4" field.
For the 16-bit variant: is the slice index offset, in the range 0 to 7, encoded in the "imm3" field.
For the 32-bit variant: is the slice index offset, in the range 0 to 3, encoded in the "imm2" field.
For the 64-bit variant: is the slice index offset, in the range 0 to 1, encoded in the "i1" field.
For the 128-bit variant: is the slice index offset 0.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

The description of MOVA (vector to tile) gives the operational pseudocode for this instruction.

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

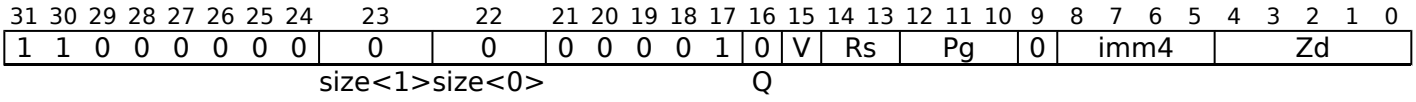
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MOVA (tile to vector)

Move ZA tile slice to vector register

The instruction operates on individual horizontal or vertical slices within a named ZA tile of the specified element size. The slice number within the tile is selected by the sum of the slice index register and an immediate, modulo the number of such elements in a vector. The immediate is in the range 0 to the number of elements in a 128-bit vector segment minus 1.
Inactive elements in the destination vector remain unmodified.
This instruction is used by the alias MOV (tile to vector).
It has encodings from 5 classes: 8-bit , 16-bit , 32-bit , 64-bit and 128-bit

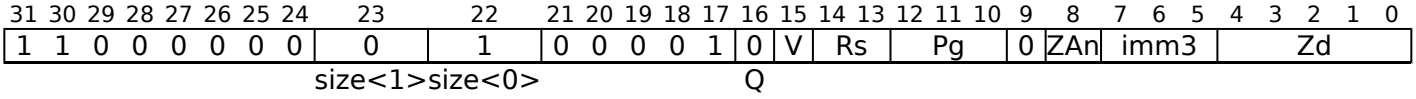
8-bit
(FEAT_SME)



MOVA <Zd>.B, <Pg>/M, ZA0<HV>.B[<Ws>, <imm>]

```
if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = 0;
integer imm = UInt(imm4);
constant integer esize = 8;
integer d = UInt(Zd);
boolean vertical = V == '1';
```

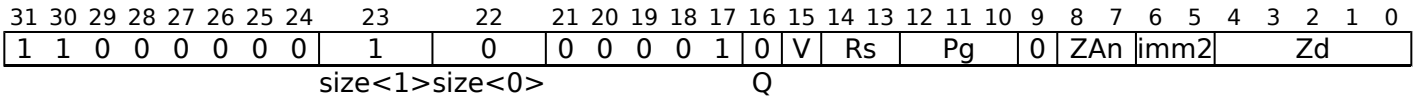
16-bit
(FEAT_SME)



MOVA <Zd>.H, <Pg>/M, <ZAn><HV>.H[<Ws>, <imm>]

```
if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(ZAn);
integer imm = UInt(imm3);
constant integer esize = 16;
integer d = UInt(Zd);
boolean vertical = V == '1';
```

32-bit
(FEAT_SME)

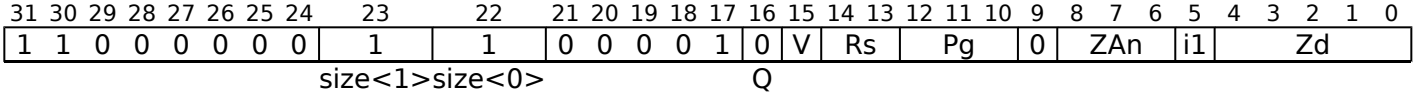


MOVA <Zd>.S, <Pg>/M, <ZAn><HV>.S[<Ws>, <imm>]

```
if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(ZAn);
integer imm = UInt(imm2);
constant integer esize = 32;
integer d = UInt(Zd);
boolean vertical = V == '1';
```

64-bit

(FEAT_SME)

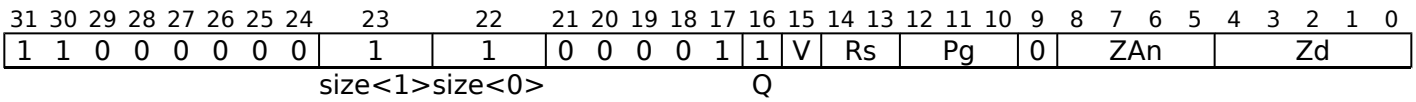


MOVA <Zd>.D, <Pg>/M, <ZAn><HV>.D[<Ws>, <imm>]

```
if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(ZAn);
integer imm = UInt(i1);
constant integer esize = 64;
integer d = UInt(Zd);
boolean vertical = V == '1';
```

128-bit

(FEAT_SME)



MOVA <Zd>.Q, <Pg>/M, <ZAn><HV>.Q[<Ws>, <imm>]

```
if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(ZAn);
integer imm = 0;
constant integer esize = 128;
integer d = UInt(Zd);
boolean vertical = V == '1';
```

Assembler Symbols

<Zd> Is the name of the destination scalable vector register, encoded in the "Zd" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<ZAn> For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAn" field.
For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAn" field.
For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAn" field.
For the 128-bit variant: is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the "ZAn" field.

<HV> Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<imm> For the 8-bit variant: is the slice index offset, in the range 0 to 15, encoded in the "imm4" field.

For the 16-bit variant: is the slice index offset, in the range 0 to 7, encoded in the "imm3" field.

For the 32-bit variant: is the slice index offset, in the range 0 to 3, encoded in the "imm2" field.

For the 64-bit variant: is the slice index offset, in the range 0 to 1, encoded in the "i1" field.

For the 128-bit variant: is the slice index offset 0.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(32) index = X[s, 32];
integer  slice = (UInt(index) + imm) MOD dim;
bits(VL) operand = ZAslice[n, esize, vertical, slice, VL];
bits(VL) result = Z[d, VL];

for e = 0 to dim-1
    bits(esize) element = Elem[operand, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = element;

Z[d, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

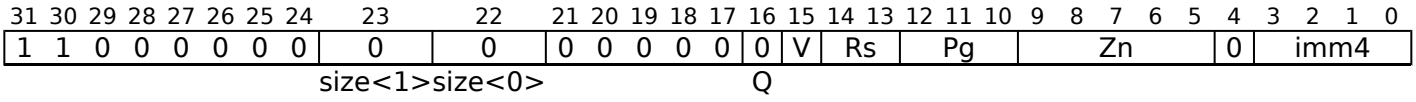
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

MOVA (vector to tile)

Move vector register to ZA tile slice

The instruction operates on individual horizontal or vertical slices within a named ZA tile of the specified element size. The slice number within the tile is selected by the sum of the slice index register and an immediate, modulo the number of such elements in a vector. The immediate is in the range 0 to the number of elements in a 128-bit vector segment minus 1.
Inactive elements in the destination slice remain unmodified.
This instruction is used by the alias MOV (vector to tile).
It has encodings from 5 classes: 8-bit , 16-bit , 32-bit , 64-bit and 128-bit

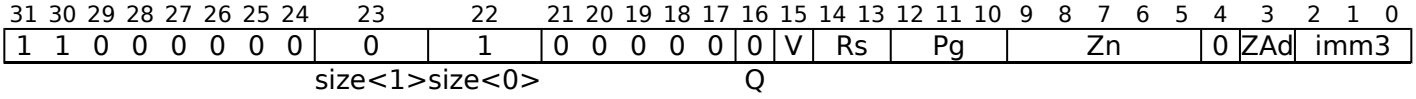
8-bit
(FEAT_SME)



MOVA ZA0<HV>.B[<Ws>, <imm>], <Pg>/M, <Zn>.B

```
if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(Zn);
integer d = 0;
integer imm = UInt(imm4);
constant integer esize = 8;
boolean vertical = V == '1';
```

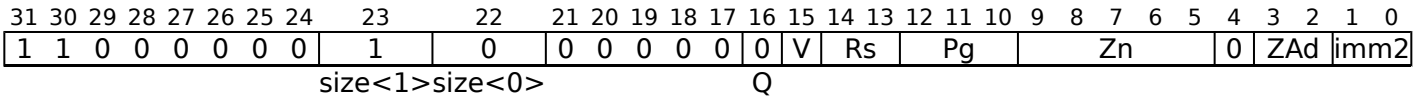
16-bit
(FEAT_SME)



MOVA <ZAd><HV>.H[<Ws>, <imm>], <Pg>/M, <Zn>.H

```
if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(Zn);
integer d = UInt(ZAd);
integer imm = UInt(imm3);
constant integer esize = 16;
boolean vertical = V == '1';
```

32-bit
(FEAT_SME)

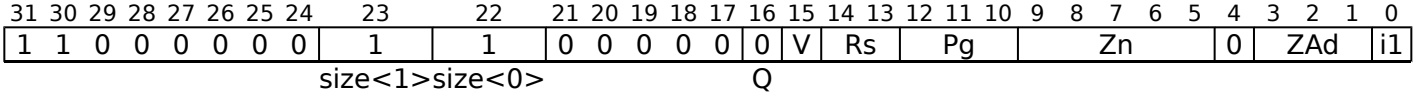


MOVA <ZAd><HV>.S[<Ws>, <imm>], <Pg>/M, <Zn>.S

```
if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(Zn);
integer d = UInt(ZAd);
integer imm = UInt(imm2);
constant integer esize = 32;
boolean vertical = V == '1';
```

64-bit

(FEAT_SME)

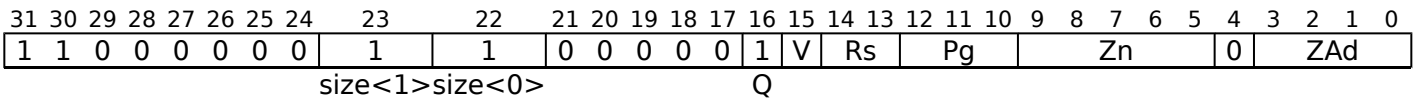


MOVA <ZAd><HV>.D[<Ws>, <imm>], <Pg>/M, <Zn>.D

```
if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(Zn);
integer d = UInt(ZAd);
integer imm = UInt(i1);
constant integer esize = 64;
boolean vertical = V == '1';
```

128-bit

(FEAT_SME)



MOVA <ZAd><HV>.Q[<Ws>, <imm>], <Pg>/M, <Zn>.Q

```
if !HaveSME() then UNDEFINED;
integer g = UInt(Pg);
integer s = UInt('011':Rs);
integer n = UInt(Zn);
integer d = UInt(ZAd);
integer imm = 0;
constant integer esize = 128;
boolean vertical = V == '1';
```

Assembler Symbols

<ZAd> For the 16-bit variant: is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAd" field.
For the 32-bit variant: is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAd" field.
For the 64-bit variant: is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAd" field.
For the 128-bit variant: is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the "ZAd" field.

<HV> Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<imm> For the 8-bit variant: is the slice index offset, in the range 0 to 15, encoded in the "imm4" field.

For the 16-bit variant: is the slice index offset, in the range 0 to 7, encoded in the "imm3" field.

For the 32-bit variant: is the slice index offset, in the range 0 to 3, encoded in the "imm2" field.

For the 64-bit variant: is the slice index offset, in the range 0 to 1, encoded in the "i1" field.

For the 128-bit variant: is the slice index offset 0.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Zn> Is the name of the source scalable vector register, encoded in the "Zn" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask = P[g, PL];
bits(VL) operand = Z[n, VL];
bits(32) index = X[s, 32];
integer  slice = (UInt(index) + imm) MOD dim;
bits(VL) result = ZAslice[d, esize, vertical, slice, VL];

for e = 0 to dim-1
    bits(esize) element = Elem[operand, e, esize];
    if ElemP[mask, e, esize] == '1' then
        Elem[result, e, esize] = element;

ZAslice[d, esize, vertical, slice, VL] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate register contains the same value for each execution.
  - The values of the NZCV flags.

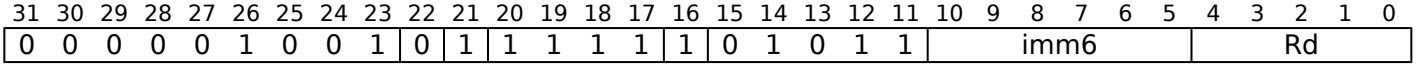
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

RDSVL

Read multiple of Streaming SVE vector register size to scalar register

Multiply the Streaming SVE vector register size in bytes by an immediate in the range -32 to 31 and place the result in the 64-bit destination general-purpose register.
This instruction does not require the PE to be in Streaming SVE mode.

SME
(FEAT_SME)



RDSVL <Xd>, #<imm>

```
if !HaveSME() then UNDEFINED;
integer d = UInt(Rd);
integer imm = SInt(imm6);
```

Assembler Symbols

<Xd>	Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.
<imm>	Is the signed immediate operand, in the range -32 to 31, encoded in the "imm6" field.

Operation

```
CheckSMEEnabled();
constant integer svl = SVL;
integer len = imm * (svl DIV 8);
X[d, 64] = len<63:0>;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

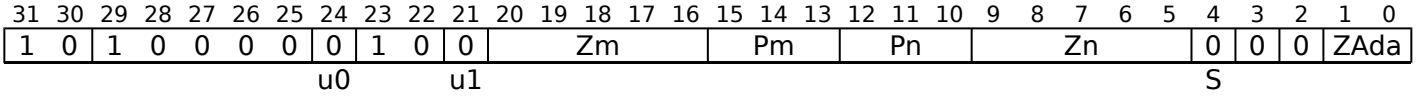
SMOPA

Signed integer sum of outer products and accumulate

The 8-bit integer variant works with a 32-bit element ZA tile.
The 16-bit integer variant works with a 64-bit element ZA tile.
The signed integer sum of outer products and accumulate instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of signed 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of signed 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of signed 16-bit integer values, and the second source holds $4 \times SVL_D$ sub-matrix of signed 16-bit integer values.
Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.
The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ widened 64-bit integer sum of outer products is then destructively added to the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and accumulate to each of the destination tile elements.
In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_D \times 4$ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_D$ sub-matrix.
ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.
It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SME)

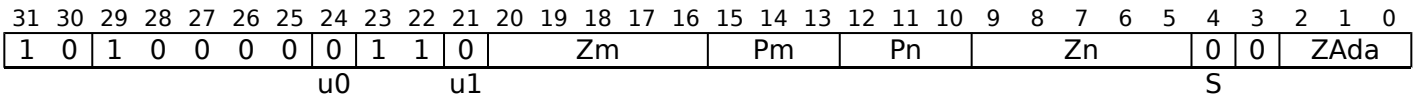


SMOPA <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B

```
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean op1_unsigned = FALSE;
boolean op2_unsigned = FALSE;
```

64-bit

(FEAT_SME_I16I64)



SMOPA <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

```
if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean op1_unsigned = FALSE;
boolean op2_unsigned = FALSE;
```

Assembler Symbols

<ZAda>	For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field. For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.
<Pn>	Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
<Pm>	Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer  prod;

for row = 0 to dim-1
    for col = 0 to dim-1
        bits(esize) sum = Elem[operand3, row*dim+col, esize];
        for k = 0 to 3
            if ElemP[mask1, 4*row + k, esize DIV 4] == '1' &&
                    ElemP[mask2, 4*col + k, esize DIV 4] == '1' then
                prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
                        Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
                if sub_op then prod = -prod;
                sum = sum + prod;

        Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.

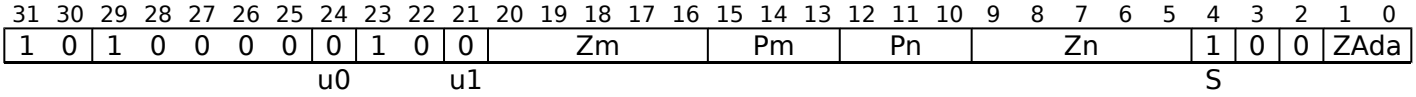
SMOPS

Signed integer sum of outer products and subtract

The 8-bit integer variant works with a 32-bit element ZA tile.
The 16-bit integer variant works with a 64-bit element ZA tile.
The signed integer sum of outer products and subtract instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of signed 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of signed 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of signed 16-bit integer values, and the second source holds $4 \times SVL_D$ sub-matrix of signed 16-bit integer values.
Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.
The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ widened 64-bit integer sum of outer products is then destructively subtracted from the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and subtract from each of the destination tile elements.
In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_D \times 4$ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_D$ sub-matrix.
ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.
It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SME)

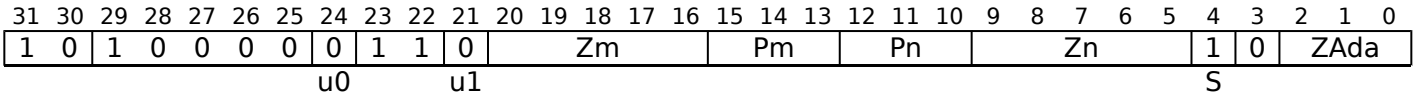


SMOPS <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B

```
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean op1_unsigned = FALSE;
boolean op2_unsigned = FALSE;
```

64-bit

(FEAT_SME_I16I64)



SMOPS <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

```
if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean op1_unsigned = FALSE;
boolean op2_unsigned = FALSE;
```

Assembler Symbols

<ZAda>	For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field. For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.
<Pn>	Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
<Pm>	Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer  prod;

for row = 0 to dim-1
    for col = 0 to dim-1
        bits(esize) sum = Elem[operand3, row*dim+col, esize];
        for k = 0 to 3
            if ElemP[mask1, 4*row + k, esize DIV 4] == '1' &&
                    ElemP[mask2, 4*col + k, esize DIV 4] == '1' then
                prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
                        Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
                if sub_op then prod = -prod;
                sum = sum + prod;

        Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

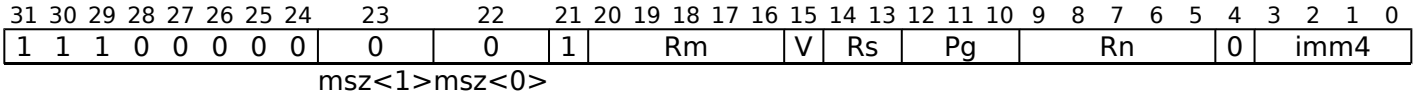
- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.

ST1B

Contiguous store of bytes from 8-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and an immediate, modulo the number of 8-bit elements in a vector. The immediate is in the range 0 to 15. The memory address is generated by scalar base and optional scalar offset which is added to the base address. Inactive elements are not written to memory.

SME
(FEAT_SME)



ST1B { ZA0<HV>.B[<Ws>, <imm>] }, <Pg>, [<Xn|SP>{, <Xm>}]

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = 0;
integer imm = UInt(imm4);
constant integer esize = 8;
boolean vertical = V == '1';
```

Assembler Symbols

<HV> Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<imm> Is the slice index offset, in the range 0 to 15, encoded in the "imm4" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g, PL];
bits(64) offset = X[m, 64];
bits(32) index = X[s, 32];
integer  slice = (UInt(index) + imm) MOD dim;
bits(VL) src;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
    if AnyActiveElement(mask, esize) ||
          ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

src = ZAslice[t, esize, vertical, slice, VL];
for e = 0 to dim-1
    addr = base + UInt(offset) * mbytes;
    if ElemP[mask, e, esize] == '1' then
        Mem[addr, mbytes, AccType_SME] = Elem[src, e, esize];
    offset = offset + 1;
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

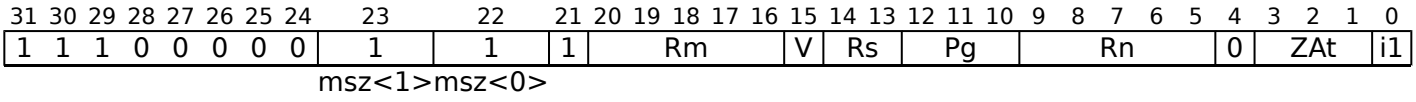
ST1D

Contiguous store of doublewords from 64-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and an immediate, modulo the number of 64-bit elements in a vector. The immediate is in the range 0 to 1. The memory address is generated by scalar base and optional scalar offset which is multiplied by 8 and added to the base address. Inactive elements are not written to memory.

SME

(FEAT_SME)



ST1D { <ZAt><HV>.D[<Ws>, <imm>] }, <Pg>, [<Xn|SP>{, <Xm>, LSL #3}]

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = UInt(ZAt);
integer imm = UInt(i1);
constant integer esize = 64;
boolean vertical = V == '1';
```

Assembler Symbols

<ZAt> Is the name of the ZA tile ZA0-ZA7 to be accessed, encoded in the "ZAt" field.

<HV> Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<imm> Is the slice index offset, in the range 0 to 1, encoded in the "i1" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g, PL];
bits(64) offset = X[m, 64];
bits(32) index = X[s, 32];
integer  slice = (UInt(index) + imm) MOD dim;
bits(VL) src;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
    if AnyActiveElement(mask, esize) ||
          ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

src = ZAslice[t, esize, vertical, slice, VL];
for e = 0 to dim-1
    addr = base + UInt(offset) * mbytes;
    if ElemP[mask, e, esize] == '1' then
        Mem[addr, mbytes, AccType_SME] = Elem[src, e, esize];
    offset = offset + 1;
```

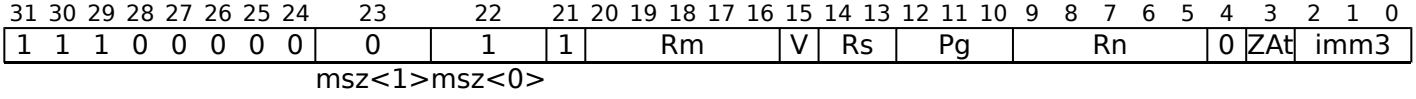
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST1H

Contiguous store of halfwords from 16-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and an immediate, modulo the number of 16-bit elements in a vector. The immediate is in the range 0 to 7. The memory address is generated by scalar base and optional scalar offset which is multiplied by 2 and added to the base address. Inactive elements are not written to memory.

SME
(FEAT_SME)



ST1H { <ZAt><HV>.H[<Ws>, <imm>] }, <Pg>, [<Xn|SP>{, <Xm>, LSL #1}]

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = UInt(ZAt);
integer imm = UInt(imm3);
constant integer esize = 16;
boolean vertical = V == '1';
```

Assembler Symbols

<ZAt> Is the name of the ZA tile ZA0-ZA1 to be accessed, encoded in the "ZAt" field.

<HV> Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<imm> Is the slice index offset, in the range 0 to 7, encoded in the "imm3" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g, PL];
bits(64) offset = X[m, 64];
bits(32) index = X[s, 32];
integer  slice = (UInt(index) + imm) MOD dim;
bits(VL) src;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
    if AnyActiveElement(mask, esize) ||
          ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

src = ZAslice[t, esize, vertical, slice, VL];
for e = 0 to dim-1
    addr = base + UInt(offset) * mbytes;
    if ElemP[mask, e, esize] == '1' then
        Mem[addr, mbytes, AccType_SME] = Elem[src, e, esize];
    offset = offset + 1;
```

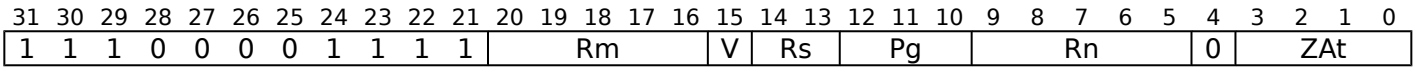
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST1Q

Contiguous store of quadwords from 128-bit element ZA tile slice

The slice number in the tile is selected by the slice index register, modulo the number of 128-bit elements in a Streaming SVE vector. The memory address is generated by scalar base and optional scalar offset which is multiplied by 16 and added to the base address. Inactive elements are not written to memory.

SME
(FEAT_SME)



ST1Q { <ZAt><HV>.Q[<Ws>, <imm>] }, <Pg>, [<Xn|SP>{, <Xm>, LSL #4}]

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = UInt(ZAt);
integer imm = 0;
constant integer esize = 128;
boolean vertical = V == '1';
```

Assembler Symbols

<ZAt> Is the name of the ZA tile ZA0-ZA15 to be accessed, encoded in the "ZAt" field.

<HV> Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<imm> Is the slice index offset 0.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g, PL];
bits(64) offset = X[m, 64];
bits(32) index = X[s, 32];
integer  slice = (UInt(index) + imm) MOD dim;
bits(VL) src;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
    if AnyActiveElement(mask, esize) ||
          ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

src = ZAslice[t, esize, vertical, slice, VL];
for e = 0 to dim-1
    addr = base + UInt(offset) * mbytes;
    if ElemP[mask, e, esize] == '1' then
        Mem[addr, mbytes, AccType_SME] = Elem[src, e, esize];
    offset = offset + 1;
```

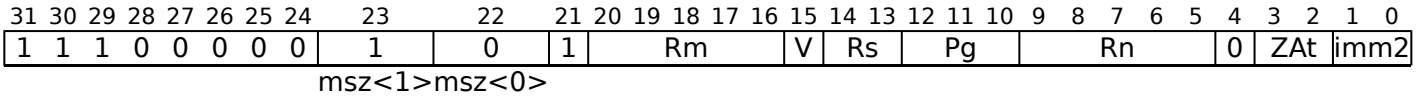
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

ST1W

Contiguous store of words from 32-bit element ZA tile slice

The slice number within the tile is selected by the sum of the slice index register and an immediate, modulo the number of 32-bit elements in a vector. The immediate is in the range 0 to 3. The memory address is generated by scalar base and optional scalar offset which is multiplied by 4 and added to the base address. Inactive elements are not written to memory.

SME
(FEAT_SME)



ST1W { <ZAt><HV>.S[<Ws>, <imm>] }, <Pg>, [<Xn|SP>{, <Xm>, LSL #2}]

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer m = UInt(Rm);
integer g = UInt('0':Pg);
integer s = UInt('011':Rs);
integer t = UInt(ZAt);
integer imm = UInt(imm2);
constant integer esize = 32;
boolean vertical = V == '1';
```

Assembler Symbols

<ZAt> Is the name of the ZA tile ZA0-ZA3 to be accessed, encoded in the "ZAt" field.

<HV> Is the horizontal or vertical slice indicator, encoded in “V”:

V	<HV>
0	H
1	V

<Ws> Is the 32-bit name of the slice index register W12-W15, encoded in the "Rs" field.

<imm> Is the slice index offset, in the range 0 to 3, encoded in the "imm2" field.

<Pg> Is the name of the governing scalable predicate register P0-P7, encoded in the "Pg" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Xm> Is the optional 64-bit name of the general-purpose offset register, defaulting to XZR, encoded in the "Rm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(64) base;
bits(64) addr;
bits(PL) mask = P[g, PL];
bits(64) offset = X[m, 64];
bits(32) index = X[s, 32];
integer  slice = (UInt(index) + imm) MOD dim;
bits(VL) src;
constant integer mbytes = esize DIV 8;

if HaveMTEExt() then SetTagCheckedInstruction(TRUE);

if n == 31 then
    if AnyActiveElement(mask, esize) ||
          ConstrainUnpredictableBool(Unpredictable_CHECKSPNONEACTIVE) then
        CheckSPAlignment();
    base = SP[];
else
    base = X[n, 64];

src = ZAslice[t, esize, vertical, slice, VL];
for e = 0 to dim-1
    addr = base + UInt(offset) * mbytes;
    if ElemP[mask, e, esize] == '1' then
        Mem[addr, mbytes, AccType_SME] = Elem[src, e, esize];
    offset = offset + 1;
```

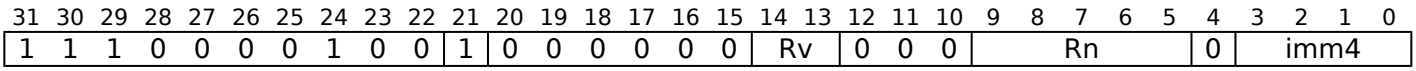
Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

STR

Store vector from ZA array

The ZA array vector is selected by the sum of the vector select register and an immediate, modulo the number of bytes in a Streaming SVE vector. The immediate is in the range 0 to 15. The memory address is generated by scalar base, plus the same optional immediate offset multiplied by the current vector length in bytes. This instruction is unpredicated.
The store is performed as contiguous byte accesses, with no endian conversion and no guarantee of single-copy atomicity larger than a byte. However, if alignment is checked, then the base register must be aligned to 16 bytes.
This instruction does not require the PE to be in Streaming SVE mode, and it is expected that this instruction will not experience a significant slowdown due to contention with other PEs that are executing in Streaming SVE mode.

SME
(FEAT_SME)



STR ZA[<Wv>, <imm>], [<Xn|SP>{, #<imm>, MUL VL}]

```
if !HaveSME() then UNDEFINED;
integer n = UInt(Rn);
integer v = UInt('011':Rv);
integer imm = UInt(imm4);
```

Assembler Symbols

<Wv>	Is the 32-bit name of the vector select register W12-W15, encoded in the "Rv" field.
<imm>	Is the vector select offset and optional memory offset, in the range 0 to 15, defaulting to 0, encoded in the "imm4" field.
<Xn|SP>	Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckSMEAndZAEnabled();
constant integer svl = SVL;
constant integer dim = svl DIV 8;
bits(32)  idx = X[v, 32];
integer   vec = (UInt(idx) + imm) MOD dim;
bits(svl) src;
bits(64)  base;
integer   offset = imm * dim;

if HaveTME() && TSTATE.depth > 0 then
    FailTransaction(TMFailure_ERR, FALSE);

if n == 31 then
    if HaveMTEExt() then SetTagCheckedInstruction(FALSE);
    CheckSPAlignment();
    base = SP[];
else
    if HaveMTEExt() then SetTagCheckedInstruction(TRUE);
    base = X[n, 64];

src = ZAvector[vec, svl];
boolean aligned = AArch64.CheckAlignment(base + offset, 16, AccType_SME, TRUE);
for e = 0 to dim-1
    AArch64.MemSingle[base + offset, 1, AccType_SME, aligned] = Elem[src, e, 8];
    offset = offset + 1;
```

SUMOPA

Signed by unsigned integer sum of outer products and accumulate

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The signed by unsigned integer sum of outer products and accumulate instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of signed 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of unsigned 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of signed 16-bit integer values, and the second source holds $4 \times SVL_D$ sub-matrix of unsigned 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ widened 64-bit integer sum of outer products is then destructively added to the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and accumulate to each of the destination tile elements.

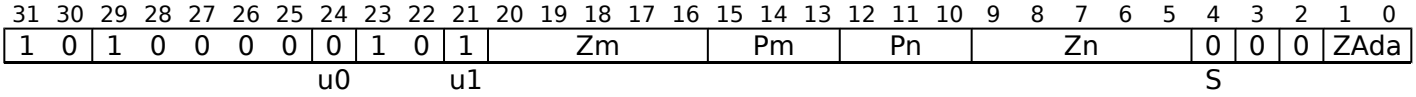
In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_D \times 4$ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_D$ sub-matrix.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SME)

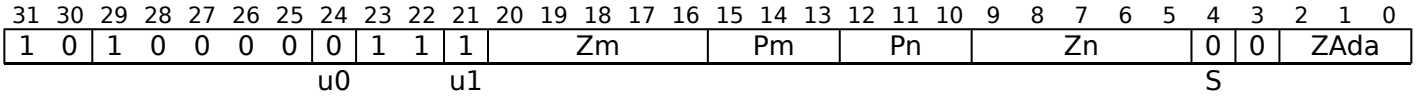


SUMOPA <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B

```
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean op1_unsigned = FALSE;
boolean op2_unsigned = TRUE;
```

64-bit

(FEAT_SME_I16I64)



SUMOPA <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

```
if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean op1_unsigned = FALSE;
boolean op2_unsigned = TRUE;
```

Assembler Symbols

<ZAda>	For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field. For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.
<Pn>	Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
<Pm>	Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer  prod;

for row = 0 to dim-1
    for col = 0 to dim-1
        bits(esize) sum = Elem[operand3, row*dim+col, esize];
        for k = 0 to 3
            if ElemP[mask1, 4*row + k, esize DIV 4] == '1' &&
                    ElemP[mask2, 4*col + k, esize DIV 4] == '1' then
                prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
                        Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
                if sub_op then prod = -prod;
                sum = sum + prod;

        Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.

SUMOPS

Signed by unsigned integer sum of outer products and subtract

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The signed by unsigned integer sum of outer products and subtract instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of signed 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of unsigned 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of signed 16-bit integer values, and the second source holds $4 \times SVL_D$ sub-matrix of unsigned 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ widened 64-bit integer sum of outer products is then destructively subtracted from the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and subtract from each of the destination tile elements.

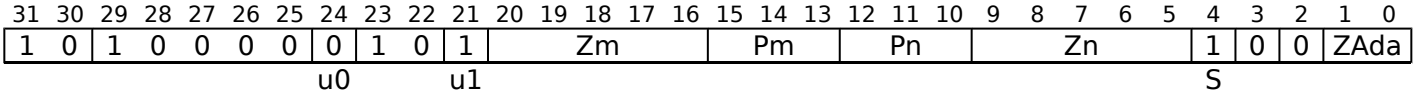
In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_D \times 4$ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_D$ sub-matrix.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SME)

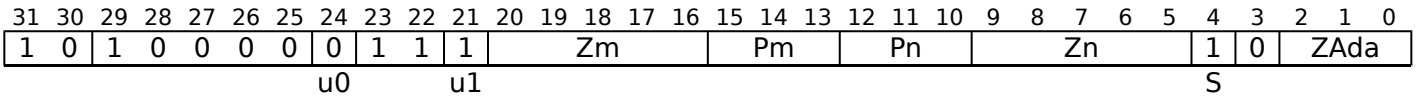


SUMOPS <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B

```
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean op1_unsigned = FALSE;
boolean op2_unsigned = TRUE;
```

64-bit

(FEAT_SME_I16I64)



SUMOPS <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

```
if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean op1_unsigned = FALSE;
boolean op2_unsigned = TRUE;
```

Assembler Symbols

<ZAda>	For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field. For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.
<Pn>	Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
<Pm>	Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer  prod;

for row = 0 to dim-1
    for col = 0 to dim-1
        bits(esize) sum = Elem[operand3, row*dim+col, esize];
        for k = 0 to 3
            if ElemP[mask1, 4*row + k, esize DIV 4] == '1' &&
                    ElemP[mask2, 4*col + k, esize DIV 4] == '1' then
                prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
                        Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
                if sub_op then prod = -prod;
                sum = sum + prod;

        Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.

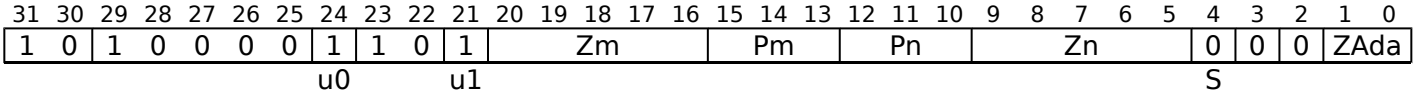
UMOPA

Unsigned integer sum of outer products and accumulate

The 8-bit integer variant works with a 32-bit element ZA tile.
The 16-bit integer variant works with a 64-bit element ZA tile.
The unsigned integer sum of outer products and accumulate instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of unsigned 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of unsigned 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of unsigned 16-bit integer values, and the second source holds $4 \times SVL_D$ sub-matrix of unsigned 16-bit integer values.
Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.
The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ widened 64-bit integer sum of outer products is then destructively added to the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and accumulate to each of the destination tile elements.
In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_D \times 4$ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_D$ sub-matrix.
ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.
It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SME)

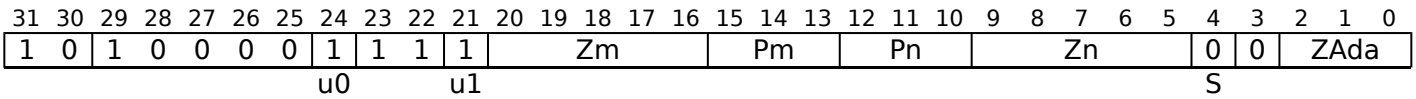


UMOPA <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B

```
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = TRUE;
```

64-bit

(FEAT_SME_I16I64)



UMOPA <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

```
if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = TRUE;
```

Assembler Symbols

<ZAda>	For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field. For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.
<Pn>	Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
<Pm>	Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer  prod;

for row = 0 to dim-1
    for col = 0 to dim-1
        bits(esize) sum = Elem[operand3, row*dim+col, esize];
        for k = 0 to 3
            if ElemP[mask1, 4*row + k, esize DIV 4] == '1' &&
                    ElemP[mask2, 4*col + k, esize DIV 4] == '1' then
                prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
                        Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
                if sub_op then prod = -prod;
                sum = sum + prod;

        Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.

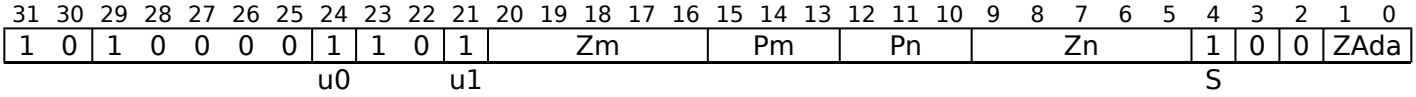
UMOPS

Unsigned integer sum of outer products and subtract

The 8-bit integer variant works with a 32-bit element ZA tile.
The 16-bit integer variant works with a 64-bit element ZA tile.
The unsigned integer sum of outer products and subtract instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of unsigned 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of unsigned 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of unsigned 16-bit integer values, and the second source holds $4 \times SVL_D$ sub-matrix of unsigned 16-bit integer values.
Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.
The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ widened 64-bit integer sum of outer products is then destructively subtracted from the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and subtract from each of the destination tile elements.
In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_D \times 4$ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_D$ sub-matrix.
ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.
It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SME)

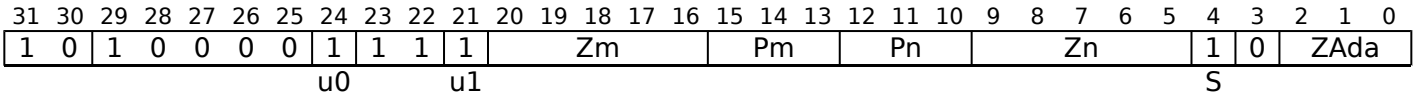


UMOPS <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B

```
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = TRUE;
```

64-bit

(FEAT_SME_I16I64)



UMOPS <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

```
if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = TRUE;
```

Assembler Symbols

<ZAda>	For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field. For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.
<Pn>	Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
<Pm>	Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer  prod;

for row = 0 to dim-1
    for col = 0 to dim-1
        bits(esize) sum = Elem[operand3, row*dim+col, esize];
        for k = 0 to 3
            if ElemP[mask1, 4*row + k, esize DIV 4] == '1' &&
                    ElemP[mask2, 4*col + k, esize DIV 4] == '1' then
                prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
                        Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
                if sub_op then prod = -prod;
                sum = sum + prod;

        Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.

USMOPA

Unsigned by signed integer sum of outer products and accumulate

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The unsigned by signed integer sum of outer products and accumulate instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of unsigned 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of signed 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of unsigned 16-bit integer values, and the second source holds $4 \times SVL_D$ sub-matrix of signed 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ widened 64-bit integer sum of outer products is then destructively added to the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and accumulate to each of the destination tile elements.

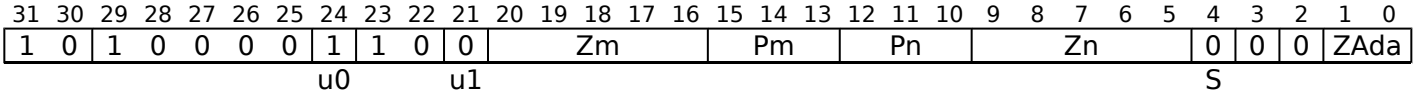
In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_D \times 4$ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_D$ sub-matrix.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SME)

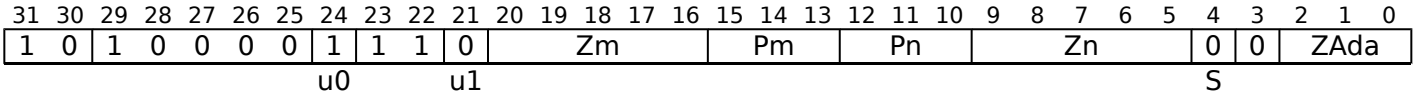


USMOPA <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B

```
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = FALSE;
```

64-bit

(FEAT_SME_I16I64)



USMOPA <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

```
if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = FALSE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = FALSE;
```

Assembler Symbols

<ZAda>	For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field. For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.
<Pn>	Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
<Pm>	Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer  prod;

for row = 0 to dim-1
    for col = 0 to dim-1
        bits(esize) sum = Elem[operand3, row*dim+col, esize];
        for k = 0 to 3
            if ElemP[mask1, 4*row + k, esize DIV 4] == '1' &&
                    ElemP[mask2, 4*col + k, esize DIV 4] == '1' then
                prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
                        Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
                if sub_op then prod = -prod;
                sum = sum + prod;

        Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.

USMOPS

Unsigned by signed integer sum of outer products and subtract

The 8-bit integer variant works with a 32-bit element ZA tile.

The 16-bit integer variant works with a 64-bit element ZA tile.

The unsigned by signed integer sum of outer products and subtract instructions multiply the sub-matrix in the first source vector by the sub-matrix in the second source vector. In case of the 8-bit integer variant, the first source holds $SVL_S \times 4$ sub-matrix of unsigned 8-bit integer values, and the second source holds $4 \times SVL_S$ sub-matrix of signed 8-bit integer values. In case of the 16-bit integer variant, the first source holds $SVL_D \times 4$ sub-matrix of unsigned 16-bit integer values, and the second source holds $4 \times SVL_D$ sub-matrix of signed 16-bit integer values.

Each source vector is independently predicated by a corresponding governing predicate. When an 8-bit source element in case of 8-bit integer variant or a 16-bit source element in case of 16-bit integer variant is Inactive, it is treated as having the value 0.

The resulting $SVL_S \times SVL_S$ widened 32-bit integer or $SVL_D \times SVL_D$ widened 64-bit integer sum of outer products is then destructively subtracted from the 32-bit integer or 64-bit integer destination tile, respectively for 8-bit integer and 16-bit integer instruction variants. This is equivalent to performing a 4-way dot product and subtract from each of the destination tile elements.

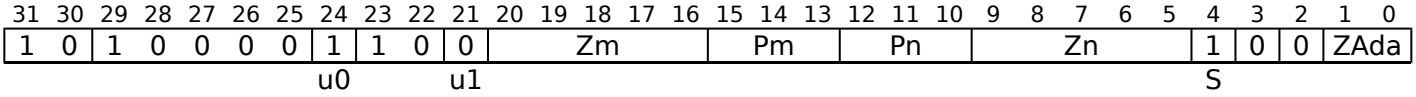
In case of the 8-bit integer variant, each 32-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_S \times 4$ sub-matrix, and each 32-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_S$ sub-matrix. In case of the 16-bit integer variant, each 64-bit container of the first source vector holds 4 consecutive column elements of each row of a $SVL_D \times 4$ sub-matrix, and each 64-bit container of the second source vector holds 4 consecutive row elements of each column of a $4 \times SVL_D$ sub-matrix.

ID_AA64SMFR0_EL1.I16I64 indicates whether the 16-bit integer variant is implemented.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

(FEAT_SME)

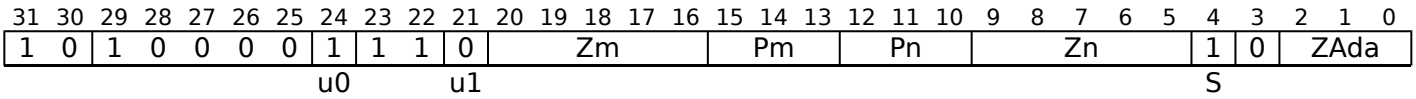


USMOPS <ZAda>.S, <Pn>/M, <Pm>/M, <Zn>.B, <Zm>.B

```
if !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = FALSE;
```

64-bit

(FEAT_SME_I16I64)



USMOPS <ZAda>.D, <Pn>/M, <Pm>/M, <Zn>.H, <Zm>.H

```
if !HaveSMEI16I64() then UNDEFINED;
constant integer esize = 64;
integer a = UInt(Pn);
integer b = UInt(Pm);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(ZAda);
boolean sub_op = TRUE;
boolean op1_unsigned = TRUE;
boolean op2_unsigned = FALSE;
```

Assembler Symbols

<ZAda>	For the 32-bit variant: is the name of the ZA tile ZA0-ZA3, encoded in the "ZAda" field. For the 64-bit variant: is the name of the ZA tile ZA0-ZA7, encoded in the "ZAda" field.
<Pn>	Is the name of the first governing scalable predicate register P0-P7, encoded in the "Pn" field.
<Pm>	Is the name of the second governing scalable predicate register P0-P7, encoded in the "Pm" field.
<Zn>	Is the name of the first source scalable vector register, encoded in the "Zn" field.
<Zm>	Is the name of the second source scalable vector register, encoded in the "Zm" field.

Operation

```
CheckStreamingSVEAndZAEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer dim = VL DIV esize;
bits(PL) mask1 = P[a, PL];
bits(PL) mask2 = P[b, PL];
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(dim*dim*esize) operand3 = ZAtile[da, esize, dim*dim*esize];
bits(dim*dim*esize) result;
integer  prod;

for row = 0 to dim-1
    for col = 0 to dim-1
        bits(esize) sum = Elem[operand3, row*dim+col, esize];
        for k = 0 to 3
            if ElemP[mask1, 4*row + k, esize DIV 4] == '1' &&
                    ElemP[mask2, 4*col + k, esize DIV 4] == '1' then
                prod = (Int(Elem[operand1, 4*row + k, esize DIV 4], op1_unsigned) *
                        Int(Elem[operand2, 4*col + k, esize DIV 4], op2_unsigned));
                if sub_op then prod = -prod;
                sum = sum + prod;

        Elem[result, row*dim+col, esize] = sum;

ZAtile[da, esize, dim*dim*esize] = result;
```

Operational information

If FEAT_SVE2 is implemented or FEAT_SME is implemented, then when PSTATE.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its operand registers when its governing predicate registers contain the same value for each execution.
  - The values of the NZCV flags.

ZERO

Zero a list of 64-bit element ZA tiles

Zeroes all bytes within each of the up to eight listed 64-bit element tiles named ZA0.D to ZA7.D, leaving the other 64-bit element tiles unmodified.

This instruction does not require the PE to be in Streaming SVE mode, and it is expected that this instruction will not experience a significant slowdown due to contention with other PEs that are executing in Streaming SVE mode.

For programmer convenience an assembler must also accept the names of 32-bit, 16-bit, and 8-bit element tiles which are converted into the corresponding set of 64-bit element tiles.

In accordance with the architecturally defined mapping between different element size tiles:

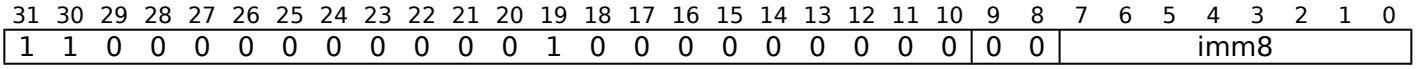
- Zeroing the 8-bit element tile name ZA0.B, or the entire array name ZA, is equivalent to zeroing all eight 64-bit element tiles named ZA0.D to ZA7.D.
- Zeroing the 16-bit element tile name ZA0.H is equivalent to zeroing 64-bit element tiles named ZA0.D, ZA2.D, ZA4.D, and ZA6.D.
- Zeroing the 16-bit element tile name ZA1.H is equivalent to zeroing 64-bit element tiles named ZA1.D, ZA3.D, ZA5.D, and ZA7.D.
- Zeroing the 32-bit element tile name ZA0.S is equivalent to zeroing 64-bit element tiles named ZA0.D and ZA4.D.
- Zeroing the 32-bit element tile name ZA1.S is equivalent to zeroing 64-bit element tiles named ZA1.D and ZA5.D.
- Zeroing the 32-bit element tile name ZA2.S is equivalent to zeroing 64-bit element tiles named ZA2.D and ZA6.D.
- Zeroing the 32-bit element tile name ZA3.S is equivalent to zeroing 64-bit element tiles named ZA3.D and ZA7.D.

The preferred disassembly of this instruction uses the shortest list of tile names that represent the encoded immediate mask.

For example:

- An immediate which encodes 64-bit element tiles ZA0.D, ZA1.D, ZA4.D, and ZA5.D is disassembled as {ZA0.S, ZA1.S}.
- An immediate which encodes 64-bit element tiles ZA0.D, ZA2.D, ZA4.D, and ZA6.D is disassembled as {ZA0.H}.
- An all-ones immediate is disassembled as {ZA}.
- An all-zeros immediate is disassembled as an empty list { }.

SME
(FEAT_SME)



ZERO { <mask> }

```
if !HaveSME() then UNDEFINED;
bits(8) mask = imm8;
constant integer esize = 64;
```

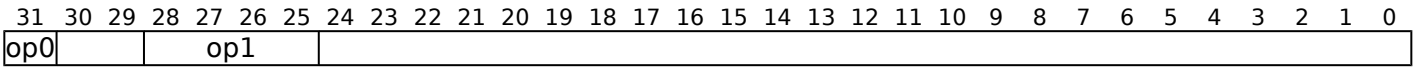
Assembler Symbols

<mask> Is a list of up to eight 64-bit element tile names separated by commas, encoded in the "imm8" field.

Operation

```
CheckSMEAndZAEnabled();
constant integer svl = SVL;
constant integer dim = svl DIV esize;
bits(dim*dim*esize) result = Zeros(dim*dim*esize);

if HaveTME() && TSTATE.depth > 0 then
    FailTransaction(TMFailure_ERR, FALSE);

for i = 0 to 7
    if mask<i> == '1' then ZAtile[i, esize, dim*dim*esize] = result;
```

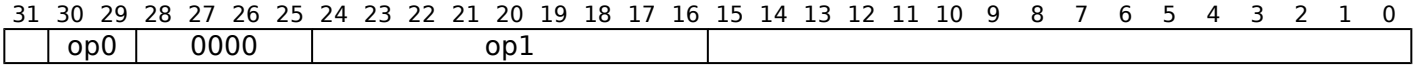
Top-level encodings for A64



Decode fields op0	op1	Instruction details
0	0000	Reserved
1	0000	SME encodings
	0001	UNALLOCATED
	0010	SVE encodings
	0011	UNALLOCATED
	100x	Data Processing -- Immediate
	101x	Branches, Exception Generating and System instructions
	x1x0	Loads and Stores
	x101	Data Processing -- Register
	x111	Data Processing -- Scalar Floating-Point and Advanced SIMD

Reserved

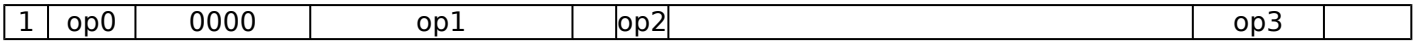
These instructions are under the top-level.



op0	Decode fields op1	Instruction details
00	000000000	UDF
	!= 000000000	UNALLOCATED
!= 00		UNALLOCATED

SME encodings

These instructions are under the top-level.

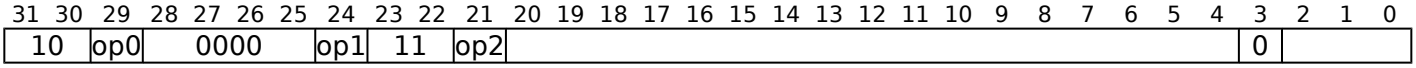


op0	Decode fields op1	op2	op3	Instruction details
0x	x0xxxx			UNALLOCATED
0x	x11xxx		x0x	SME Outer Product - 64 bit
00	x1xxxx		x1x	UNALLOCATED
00	x10xxx		x00	SME FP Outer Product - 32 bit
00	x10xxx		x01	UNALLOCATED
01	x10xxx		xx0	SME Integer Outer Product - 32 bit
01	x10xxx		xx1	UNALLOCATED
01	x11xxx		x1x	UNALLOCATED
10	0xx000	0	0xx	SME Move into Array
10	0xx000	0	1xx	UNALLOCATED
10	0xx000	1		SME Move from Array
10	0xx001			SME Misc

10	0xx010		x0x	SME Add Vector to Array
10	0xx010		x1x	UNALLOCATED
10	0xx011			UNALLOCATED
10	0xx1xx			UNALLOCATED
10	1xxxxx			UNALLOCATED
11				SME Memory

SME Outer Product - 64 bit

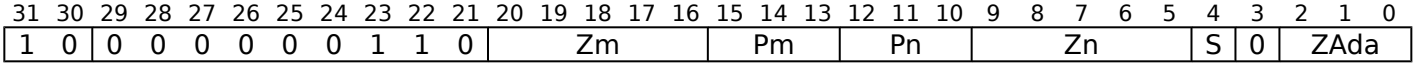
These instructions are under SME encodings.



Decode fields op0	op1	op2	Instruction details
0	0	0	SME FP64 outer product
0	0	1	UNALLOCATED
0	1		UNALLOCATED
1			SME Int16 outer product

SME FP64 outer product

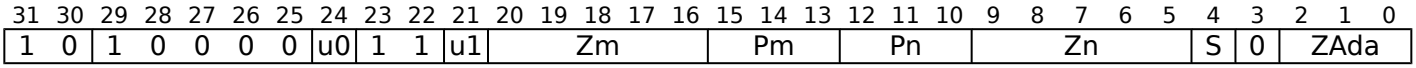
These instructions are under SME Outer Product - 64 bit.



Decode fields S	Instruction Details	Feature
0	FMOPA (non-widening)	FEAT_SME_F64F64
1	FMOPS (non-widening)	FEAT_SME_F64F64

SME Int16 outer product

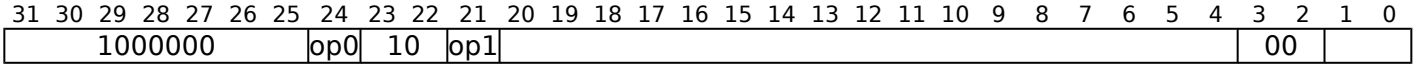
These instructions are under SME Outer Product - 64 bit.



Decode fields u0	u1	S	Instruction Details	Feature
0	0	0	SMOPA	FEAT_SME_I16I64
0	0	1	SMOPS	FEAT_SME_I16I64
0	1	0	SUMOPA	FEAT_SME_I16I64
0	1	1	SUMOPS	FEAT_SME_I16I64
1	0	0	USMOPA	FEAT_SME_I16I64
1	0	1	USMOPS	FEAT_SME_I16I64
1	1	0	UMOPA	FEAT_SME_I16I64
1	1	1	UMOPS	FEAT_SME_I16I64

SME FP Outer Product - 32 bit

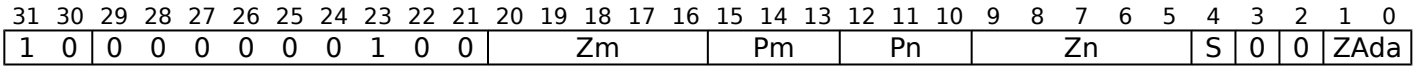
These instructions are under SME encodings.



Decode fields op0	op1	Instruction details
0	0	SME FP32 outer product
0	1	UNALLOCATED
1	0	SME widening BF16 outer product
1	1	SME FP16 widening outer product

SME FP32 outer product

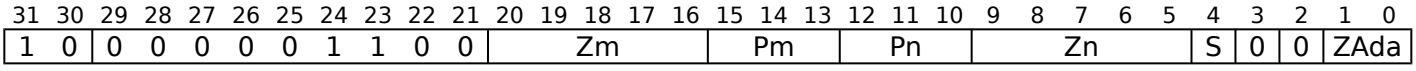
These instructions are under SME FP Outer Product - 32 bit.



Decode fields S	Instruction Details	Feature
0	FMOPA (non-widening)	FEAT_SME
1	FMOPS (non-widening)	FEAT_SME

SME widening BF16 outer product

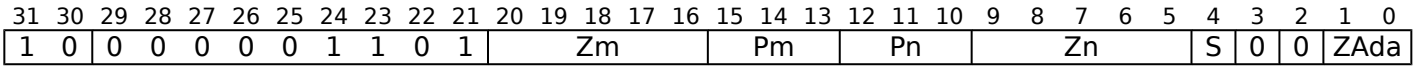
These instructions are under SME FP Outer Product - 32 bit.



Decode fields S	Instruction Details	Feature
0	BFMOPA	FEAT_SME
1	BFMOPS	FEAT_SME

SME FP16 widening outer product

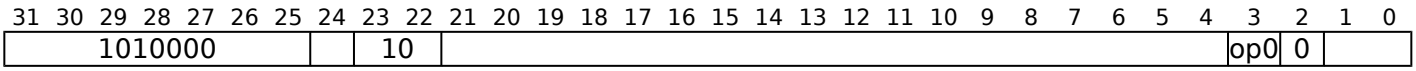
These instructions are under SME FP Outer Product - 32 bit.



Decode fields S	Instruction Details	Feature
0	FMOPA (widening)	FEAT_SME
1	FMOPS (widening)	FEAT_SME

SME Integer Outer Product - 32 bit

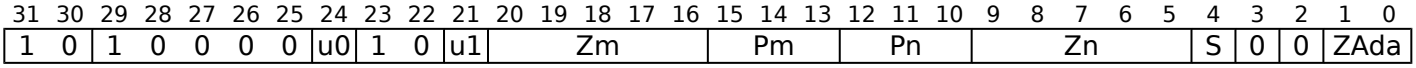
These instructions are under SME encodings.



Decode fields op0	Instruction details
0	SME Int8 outer product
1	UNALLOCATED

SME Int8 outer product

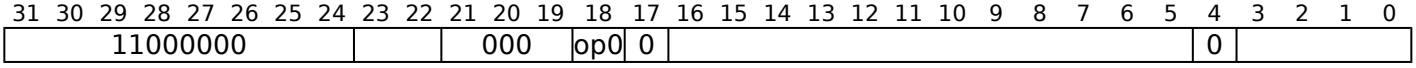
These instructions are under SME Integer Outer Product - 32 bit.



Decode fields u0	u1	S	Instruction Details	Feature
0	0	0	SMOPA	FEAT_SME
0	0	1	SMOPS	FEAT_SME
0	1	0	SUMOPA	FEAT_SME
0	1	1	SUMOPS	FEAT_SME
1	0	0	USMOPA	FEAT_SME
1	0	1	USMOPS	FEAT_SME
1	1	0	UMOPA	FEAT_SME
1	1	1	UMOPS	FEAT_SME

SME Move into Array

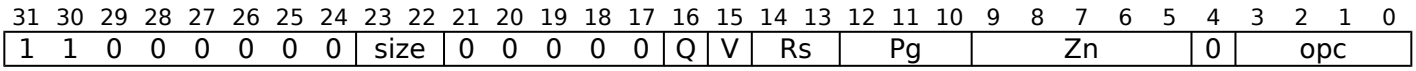
These instructions are under SME encodings.



Decode fields op0	Instruction details
0	SME move vector to array
1	UNALLOCATED

SME move vector to array

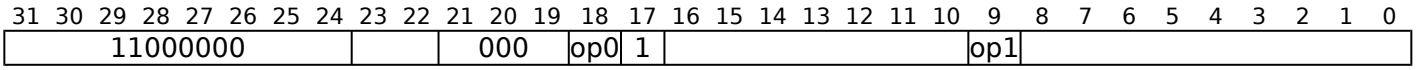
These instructions are under SME Move into Array.



Decode fields size	Q	Instruction Details	Feature
0x	1	UNALLOCATED	-
00	0	MOVA (vector to tile) — 8-bit	FEAT_SME
01	0	MOVA (vector to tile) — 16-bit	FEAT_SME
10	0	MOVA (vector to tile) — 32-bit	FEAT_SME
10	1	UNALLOCATED	-
11	0	MOVA (vector to tile) — 64-bit	FEAT_SME
11	1	MOVA (vector to tile) — 128-bit	FEAT_SME

SME Move from Array

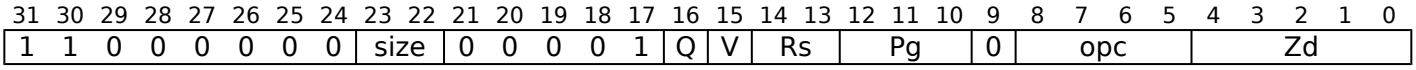
These instructions are under SME encodings.



Decode fields op0	op1	Instruction details
0	0	SME move array to vector
0	1	UNALLOCATED
1		UNALLOCATED

SME move array to vector

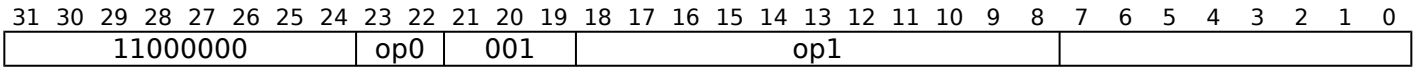
These instructions are under SME Move from Array.



Decode fields size	Q	Instruction Details	Feature
0x	1	UNALLOCATED	-
00	0	MOVA (tile to vector) — 8-bit	FEAT_SME
01	0	MOVA (tile to vector) — 16-bit	FEAT_SME
10	0	MOVA (tile to vector) — 32-bit	FEAT_SME
10	1	UNALLOCATED	-
11	0	MOVA (tile to vector) — 64-bit	FEAT_SME
11	1	MOVA (tile to vector) — 128-bit	FEAT_SME

SME Misc

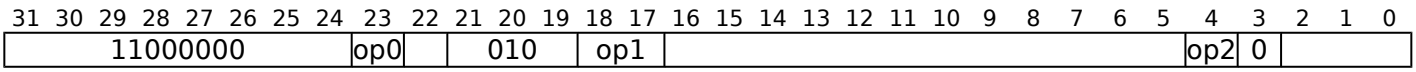
These instructions are under SME encodings.



op0	Decode fields op1	Instruction details	Architecture version
00	00000000000	ZERO	FEAT_SME
00	!= 00000000000	UNALLOCATED	-
!= 00		UNALLOCATED	-

SME Add Vector to Array

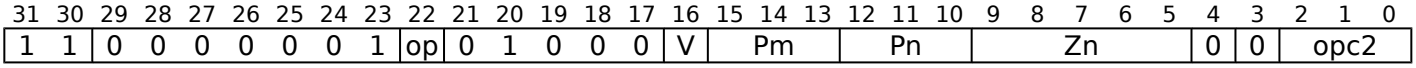
These instructions are under SME encodings.



Decode fields op0	op1	op2	Instruction details
0			UNALLOCATED
1	00	0	SME add vector to array
1	00	1	UNALLOCATED
1	!= 00		UNALLOCATED

SME add vector to array

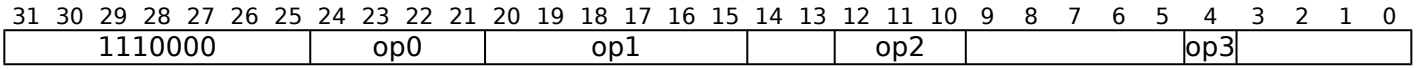
These instructions are under SME Add Vector to Array.



Decode fields op	V	opc2	Instruction Details	Feature
0		1xx	UNALLOCATED	-
0	0	0xx	ADDHA — 32-bit	FEAT_SME
0	1	0xx	ADDVA — 32-bit	FEAT_SME
1	0		ADDHA — 64-bit	FEAT_SME_I16I64
1	1		ADDVA — 64-bit	FEAT_SME_I16I64

SME Memory

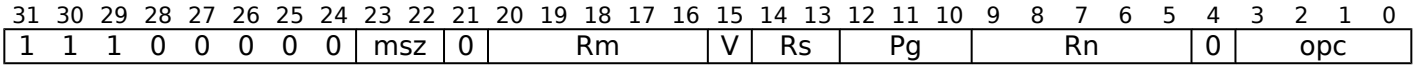
These instructions are under SME encodings.



Decode fields op0	op1	op2	op3	Instruction details	Architecture version
0xx0			0	SME load array vector (elements)	-
0xx1			0	SME store array vector (elements)	-
0xxx			1	UNALLOCATED	-
100x	000000	000	0	SME save and restore array	-
100x	000000	000	1	UNALLOCATED	-
100x	000000	!= 000		UNALLOCATED	-
100x	!= 000000			UNALLOCATED	-
101x				UNALLOCATED	-
110x				UNALLOCATED	-
1110			0	LD1Q	FEAT_SME
1111			0	ST1Q	FEAT_SME
111x			1	UNALLOCATED	-

SME load array vector (elements)

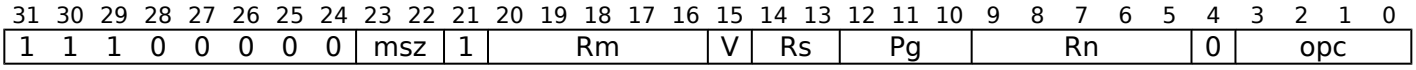
These instructions are under SME Memory.



Decode fields msz	Instruction Details	Feature
00	LD1B	FEAT_SME
01	LD1H	FEAT_SME
10	LD1W	FEAT_SME
11	LD1D	FEAT_SME

SME store array vector (elements)

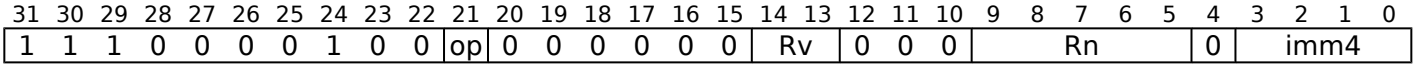
These instructions are under SME Memory.



Decode fields msz	Instruction Details	Feature
00	ST1B	FEAT_SME
01	ST1H	FEAT_SME
10	ST1W	FEAT_SME
11	ST1D	FEAT_SME

SME save and restore array

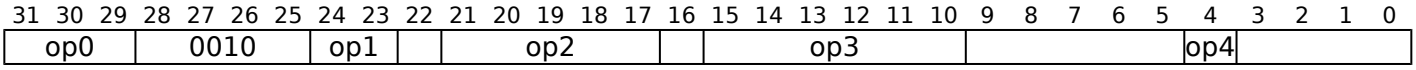
These instructions are under SME Memory.



Decode fields op	Instruction Details	Feature
0	LDR	FEAT_SME
1	STR	FEAT_SME

SVE encodings

These instructions are under the top-level.



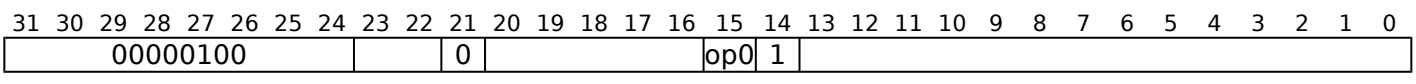
op0	op1	Decode fields op2	op3	op4	Instruction details
000	0x	0xxxx	x1xxxx		SVE Integer Multiply-Add - Predicated
000	0x	0xxxx	000xxx		SVE Integer Binary Arithmetic - Predicated
000	0x	0xxxx	001xxx		SVE Integer Reduction
000	0x	0xxxx	100xxx		SVE Bitwise Shift - Predicated
000	0x	0xxxx	101xxx		SVE Integer Unary Arithmetic - Predicated
000	0x	1xxxx	000xxx		SVE integer add/subtract vectors (unpredicated)
000	0x	1xxxx	001xxx		SVE Bitwise Logical - Unpredicated
000	0x	1xxxx	0100xx		SVE Index Generation
000	0x	1xxxx	0101xx		SVE Stack Allocation
000	0x	1xxxx	011xxx		SVE2 Integer Multiply - Unpredicated
000	0x	1xxxx	100xxx		SVE Bitwise Shift - Unpredicated
000	0x	1xxxx	1010xx		SVE address generation
000	0x	1xxxx	1011xx		SVE Integer Misc - Unpredicated
000	0x	1xxxx	11xxxx		SVE Element Count
000	1x	00xxx			SVE Bitwise Immediate
000	1x	01xxx			SVE Integer Wide Immediate - Predicated
000	1x	1xxxx	001000		DUP (indexed)
000	1x	1xxxx	001001		UNALLOCATED
000	1x	1xxxx	00101x		SVE table lookup (three sources)
000	1x	1xxxx	0011x1		UNALLOCATED
000	1x	1xxxx	001100		TBL — SVE
000	1x	1xxxx	001110		SVE Permute Vector - Unpredicated
000	1x	1xxxx	010xxx		SVE Permute Predicate

000	1x	1xxxx	011xxx		SVE permute vector elements
000	1x	1xxxx	10xxxx		SVE Permute Vector - Predicated
000	1x	1xxxx	11xxxx		SEL (vectors)
000	10	1xxxx	000xxx		SVE Permute Vector - Extract
000	11	1xxxx	000xxx		SVE Permute Vector - Segments
001	0x	0xxxx			SVE Integer Compare - Vectors
001	0x	1xxxx			SVE integer compare with unsigned immediate
001	1x	0xxxx	x0xxxx		SVE integer compare with signed immediate
001	1x	00xxx	01xxxx		SVE predicate logical operations
001	1x	00xxx	11xxxx		SVE Propagate Break
001	1x	01xxx	01xxxx		SVE Partition Break
001	1x	01xxx	11xxxx		SVE Predicate Misc
001	1x	1xxxx	00xxxx		SVE Integer Compare - Scalars
001	1x	1xxxx	01xxxx	0	SVE broadcast predicate element
001	1x	1xxxx	01xxxx	1	UNALLOCATED
001	1x	1xxxx	11xxxx		SVE Integer Wide Immediate - Unpredicated
001	1x	100xx	10xxxx		SVE Predicate Count
001	1x	101xx	1000xx		SVE Inc/Dec by Predicate Count
001	1x	101xx	1001xx		SVE Write FFR
001	1x	101xx	101xxx		UNALLOCATED
001	1x	11xxx	10xxxx		UNALLOCATED
010	0x	0xxxx	0xxxxx		SVE Integer Multiply-Add - Unpredicated
010	0x	0xxxx	10xxxx		SVE2 Integer - Predicated
010	0x	0xxxx	11000x		SVE integer clamp
010	0x	0xxxx	11001x		UNALLOCATED
010	0x	0xxxx	1101xx		UNALLOCATED
010	0x	0xxxx	111xxx		UNALLOCATED
010	0x	1xxxx			SVE Multiply - Indexed
010	1x	0xxxx	0xxxxx		SVE2 Widening Integer Arithmetic
010	1x	0xxxx	10xxxx		SVE Misc
010	1x	0xxxx	11xxxx		SVE2 Accumulate
010	1x	1xxxx	0xxxxx		SVE2 Narrowing
010	1x	1xxxx	100xxx		SVE2 character match
010	1x	1xxxx	101xxx		SVE2 Histogram Computation - Segment
010	1x	1xxxx	110xxx		HISTCNT
010	1x	1xxxx	111xxx		SVE2 Crypto Extensions
011	0x	0xxxx	0xxxxx		FCMLA (vectors)
011	0x	00x1x	1xxxxx		UNALLOCATED
011	0x	00000	100xxx		FCADD
011	0x	00000	101xxx		UNALLOCATED
011	0x	00000	11xxxx		UNALLOCATED
011	0x	00001	1xxxxx		UNALLOCATED
011	0x	0010x	100xxx		UNALLOCATED
011	0x	0010x	101xxx		SVE floating-point convert precision odd elements
011	0x	0010x	11xxxx		UNALLOCATED
011	0x	010xx	100xxx		SVE2 floating-point pairwise operations
011	0x	010xx	101xxx		UNALLOCATED
011	0x	010xx	11xxxx		UNALLOCATED
011	0x	011xx	1xxxxx		UNALLOCATED

011	0x	1xxxx	0000xx		SVE floating-point multiply-add (indexed)
011	0x	1xxxx	0001xx		SVE floating-point complex multiply-add (indexed)
011	0x	1xxxx	0010x0		SVE floating-point multiply (indexed)
011	0x	1xxxx	0010x1		UNALLOCATED
011	0x	1xxxx	0011xx		UNALLOCATED
011	0x	1xxxx	01x0xx		SVE Floating Point Widening Multiply-Add - Indexed
011	0x	1xxxx	01x1xx		UNALLOCATED
011	0x	1xxxx	10x00x		SVE Floating Point Widening Multiply-Add
011	0x	1xxxx	10x01x		UNALLOCATED
011	0x	1xxxx	10x1xx		UNALLOCATED
011	0x	1xxxx	110xxx		UNALLOCATED
011	0x	1xxxx	111000		UNALLOCATED
011	0x	1xxxx	111001		SVE floating point matrix multiply accumulate
011	0x	1xxxx	11101x		UNALLOCATED
011	0x	1xxxx	1111xx		UNALLOCATED
011	1x	0xxxx	x1xxxx		SVE floating-point compare vectors
011	1x	0xxxx	000xxx		SVE floating-point arithmetic (unpredicated)
011	1x	0xxxx	100xxx		SVE Floating Point Arithmetic - Predicated
011	1x	0xxxx	101xxx		SVE Floating Point Unary Operations - Predicated
011	1x	000xx	001xxx		SVE floating-point recursive reduction
011	1x	001xx	0010xx		UNALLOCATED
011	1x	001xx	0011xx		SVE Floating Point Unary Operations - Unpredicated
011	1x	010xx	001xxx		SVE Floating Point Compare - with Zero
011	1x	011xx	001xxx		SVE Floating Point Accumulating Reduction
011	1x	1xxxx			SVE Floating Point Multiply-Add
100					SVE Memory - 32-bit Gather and Unsized Contiguous
101					SVE Memory - Contiguous Load
110					SVE Memory - 64-bit Gather
111			0x0xxx		SVE Memory - Contiguous Store and Unsized Contiguous
111			0x1xxx		SVE Memory - Non-temporal and Multi-register Store
111			1x0xxx		SVE Memory - Scatter with Optional Sign Extend
111			101xxx		SVE Memory - Scatter
111			111xxx		SVE Memory - Contiguous Store with Immediate Offset

SVE Integer Multiply-Add - Predicated

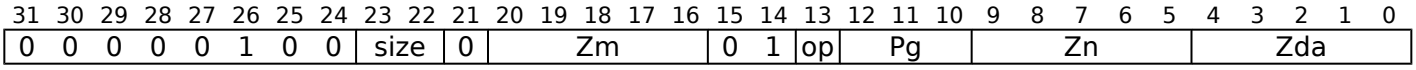
These instructions are under SVE encodings.



Decode fields op0	Instruction details
0	SVE integer multiply-accumulate writing addend (predicated)
1	SVE integer multiply-add writing multiplicand (predicated)

SVE integer multiply-accumulate writing addend (predicated)

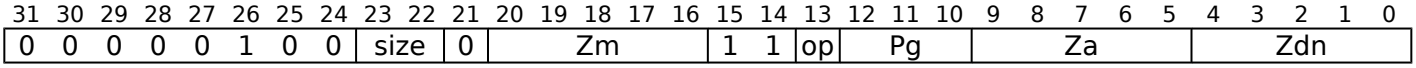
These instructions are under SVE Integer Multiply-Add - Predicated.



Decode fields op	Instruction Details
0	MLA (vectors)
1	MLS (vectors)

SVE integer multiply-add writing multiplicand (predicated)

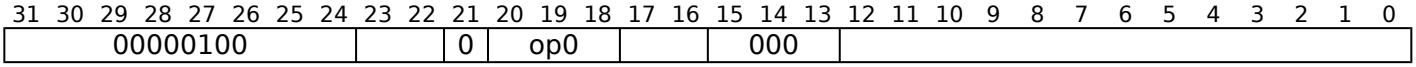
These instructions are under SVE Integer Multiply-Add - Predicated.



Decode fields op	Instruction Details
0	MAD
1	MSB

SVE Integer Binary Arithmetic - Predicated

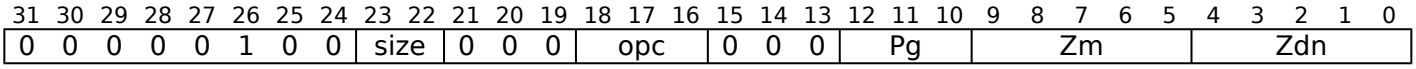
These instructions are under SVE encodings.



Decode fields op0	Instruction details
00x	SVE integer add/subtract vectors (predicated)
01x	SVE integer min/max/difference (predicated)
100	SVE integer multiply vectors (predicated)
101	SVE integer divide vectors (predicated)
11x	SVE bitwise logical operations (predicated)

SVE integer add/subtract vectors (predicated)

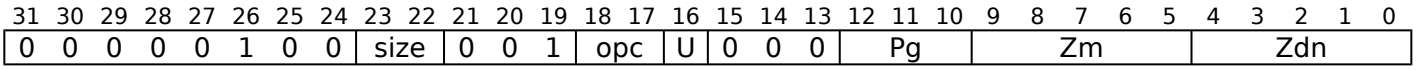
These instructions are under SVE Integer Binary Arithmetic - Predicated.



Decode fields opc	Instruction Details
000	ADD (vectors, predicated)
001	SUB (vectors, predicated)
010	UNALLOCATED
011	SUBR (vectors)
1xx	UNALLOCATED

SVE integer min/max/difference (predicated)

These instructions are under SVE Integer Binary Arithmetic - Predicated.



Decode fields opc	U	Instruction Details
00	0	SMAX (vectors)
00	1	UMAX (vectors)
01	0	SMIN (vectors)
01	1	UMIN (vectors)
10	0	SABD
10	1	UABD
11		UNALLOCATED

SVE integer multiply vectors (predicated)

These instructions are under SVE Integer Binary Arithmetic - Predicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size		0	1	0	0	H	U	0	0	0	Pg			Zm					Zdn				

Decode fields H	U	Instruction Details
0	0	MUL (vectors, predicated)
0	1	UNALLOCATED
1	0	SMULH (predicated)
1	1	UMULH (predicated)

SVE integer divide vectors (predicated)

These instructions are under SVE Integer Binary Arithmetic - Predicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size		0	1	0	1	R	U	0	0	0	Pg			Zm					Zdn				

Decode fields R	U	Instruction Details
0	0	SDIV
0	1	UDIV
1	0	SDIVR
1	1	UDIVR

SVE bitwise logical operations (predicated)

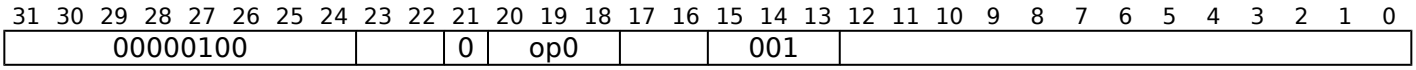
These instructions are under SVE Integer Binary Arithmetic - Predicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	size		0	1	1	opc			0	0	0	Pg			Zm					Zdn				

Decode fields opc	Instruction Details
000	ORR (vectors, predicated)
001	EOR (vectors, predicated)
010	AND (vectors, predicated)
011	BIC (vectors, predicated)
1xx	UNALLOCATED

SVE Integer Reduction

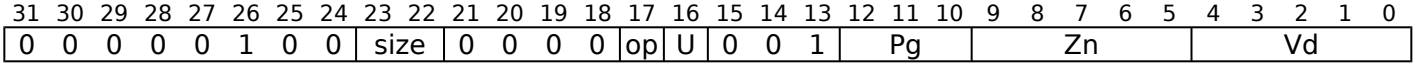
These instructions are under SVE encodings.



Decode fields op0	Instruction details
000	SVE integer add reduction (predicated)
010	SVE integer min/max reduction (predicated)
0x1	UNALLOCATED
10x	SVE constructive prefix (predicated)
110	SVE bitwise logical reduction (predicated)
111	UNALLOCATED

SVE integer add reduction (predicated)

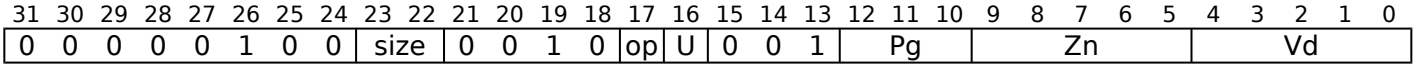
These instructions are under SVE Integer Reduction.



Decode fields op	U	Instruction Details
0	0	SADDV
0	1	UADDV
1		UNALLOCATED

SVE integer min/max reduction (predicated)

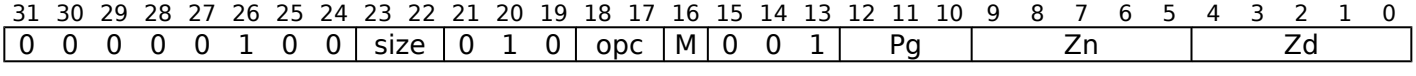
These instructions are under SVE Integer Reduction.



Decode fields op	U	Instruction Details
0	0	SMAXV
0	1	UMAXV
1	0	SMINV
1	1	UMINV

SVE constructive prefix (predicated)

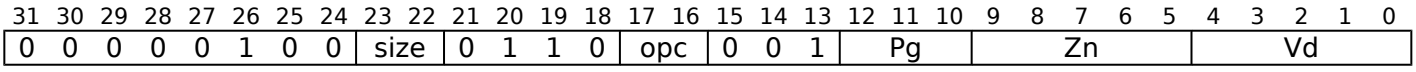
These instructions are under SVE Integer Reduction.



Decode fields opc	Instruction Details
00	MOVPRFX (predicated)
01	UNALLOCATED
1x	UNALLOCATED

SVE bitwise logical reduction (predicated)

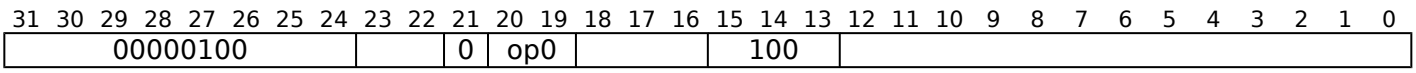
These instructions are under SVE Integer Reduction.



Decode fields opc	Instruction Details
00	ORV
01	EORV
10	ANDV
11	UNALLOCATED

SVE Bitwise Shift - Predicated

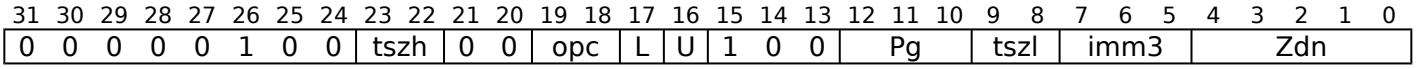
These instructions are under SVE encodings.



Decode fields op0	Instruction details
0x	SVE bitwise shift by immediate (predicated)
10	SVE bitwise shift by vector (predicated)
11	SVE bitwise shift by wide elements (predicated)

SVE bitwise shift by immediate (predicated)

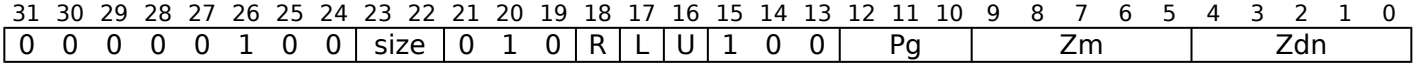
These instructions are under SVE Bitwise Shift - Predicated.



Decode fields opc	L	U	Instruction Details
00	0	0	ASR (immediate, predicated)
00	0	1	LSR (immediate, predicated)
00	1	0	UNALLOCATED
00	1	1	LSL (immediate, predicated)
01	0	0	ASRD
01	0	1	UNALLOCATED
01	1	0	SQSHL (immediate)
01	1	1	UQSHL (immediate)
10			UNALLOCATED
11	0	0	SRSHR
11	0	1	URSHR
11	1	0	UNALLOCATED
11	1	1	SQSHLU

SVE bitwise shift by vector (predicated)

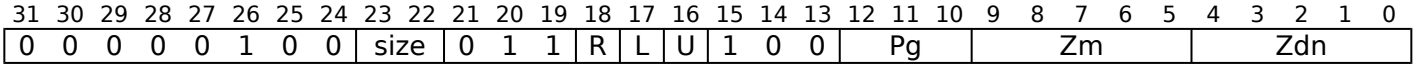
These instructions are under SVE Bitwise Shift - Predicated.



Decode fields R	L	U	Instruction Details
	1	0	UNALLOCATED
0	0	0	ASR (vectors)
0	0	1	LSR (vectors)
0	1	1	LSL (vectors)
1	0	0	ASRR
1	0	1	LSRR
1	1	1	LSLR

SVE bitwise shift by wide elements (predicated)

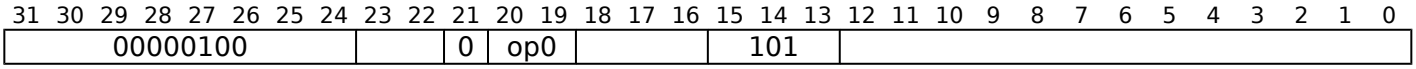
These instructions are under SVE Bitwise Shift - Predicated.



Decode fields R	L	U	Instruction Details
0	0	0	ASR (wide elements, predicated)
0	0	1	LSR (wide elements, predicated)
0	1	0	UNALLOCATED
0	1	1	LSL (wide elements, predicated)
1			UNALLOCATED

SVE Integer Unary Arithmetic - Predicated

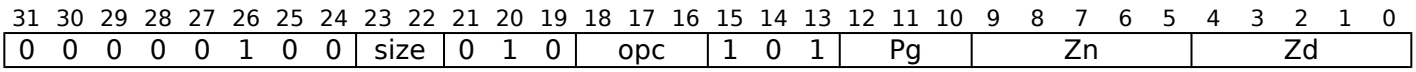
These instructions are under SVE encodings.



Decode fields op0	Instruction details
0x	UNALLOCATED
10	SVE integer unary operations (predicated)
11	SVE bitwise unary operations (predicated)

SVE integer unary operations (predicated)

These instructions are under SVE Integer Unary Arithmetic - Predicated.

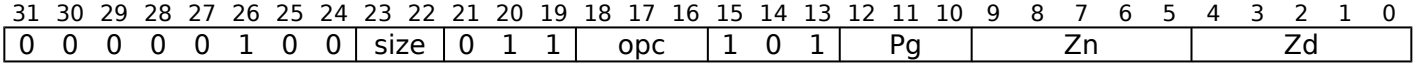


Decode fields opc	Instruction Details
000	SXTB, SXTH, SXTW — SXTB
001	UXTB, UXTH, UXTW — UXTB
010	SXTB, SXTH, SXTW — SXTH
011	UXTB, UXTH, UXTW — UXTH
100	SXTB, SXTH, SXTW — SXTW
101	UXTB, UXTH, UXTW — UXTW
110	ABS

Decode fields opc	Instruction Details
111	NEG

SVE bitwise unary operations (predicated)

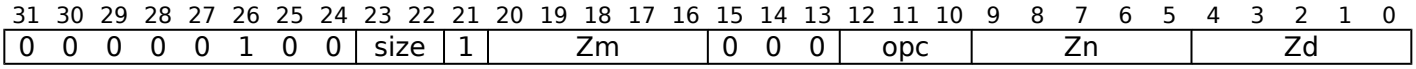
These instructions are under SVE Integer Unary Arithmetic - Predicated.



Decode fields opc	Instruction Details
000	CLS
001	CLZ
010	CNT
011	CNOT
100	FABS
101	FNEG
110	NOT (vector)
111	UNALLOCATED

SVE integer add/subtract vectors (unpredicated)

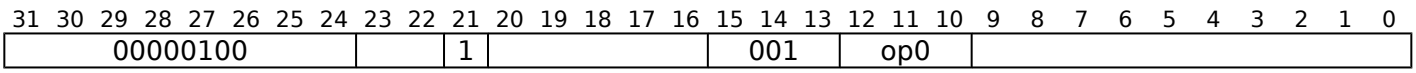
These instructions are under SVE encodings.



Decode fields opc	Instruction Details
000	ADD (vectors, unpredicated)
001	SUB (vectors, unpredicated)
01x	UNALLOCATED
100	SQADD (vectors, unpredicated)
101	UQADD (vectors, unpredicated)
110	SQSUB (vectors, unpredicated)
111	UQSUB (vectors, unpredicated)

SVE Bitwise Logical - Unpredicated

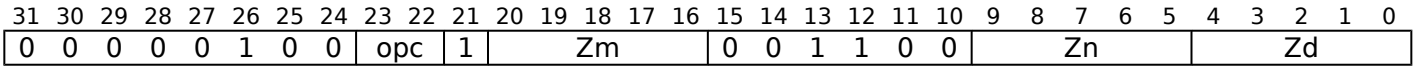
These instructions are under SVE encodings.



Decode fields op0	Instruction details
0xx	UNALLOCATED
100	SVE bitwise logical operations (unpredicated)
101	XAR
11x	SVE2 bitwise ternary operations

SVE bitwise logical operations (unpredicated)

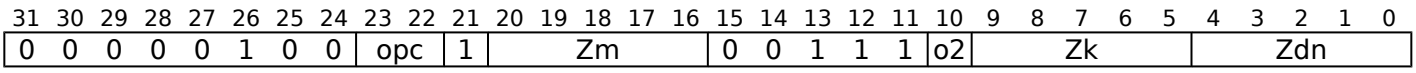
These instructions are under SVE Bitwise Logical - Unpredicated.



Decode fields opc	Instruction Details
00	AND (vectors, unpredicated)
01	ORR (vectors, unpredicated)
10	EOR (vectors, unpredicated)
11	BIC (vectors, unpredicated)

SVE2 bitwise ternary operations

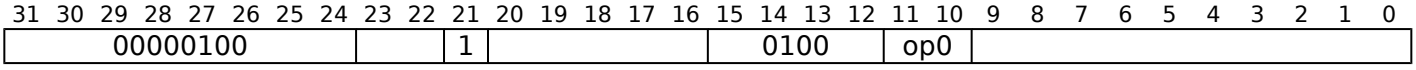
These instructions are under SVE Bitwise Logical - Unpredicated.



Decode fields opc	o2	Instruction Details
00	0	EOR3
00	1	BSL
01	0	BCAX
01	1	BSL1N
1x	0	UNALLOCATED
10	1	BSL2N
11	1	NBSL

SVE Index Generation

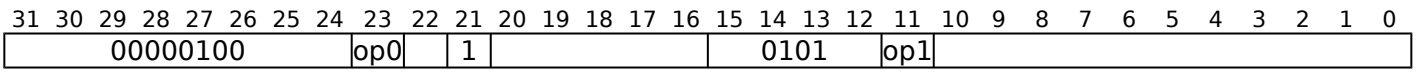
These instructions are under SVE encodings.



Decode fields op0	Instruction details
00	INDEX (immediates)
01	INDEX (scalar, immediate)
10	INDEX (immediate, scalar)
11	INDEX (scalars)

SVE Stack Allocation

These instructions are under SVE encodings.

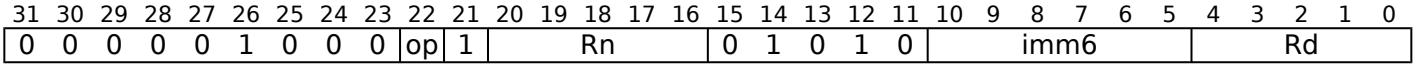


Decode fields op0	op1	Instruction details
0	0	SVE stack frame adjustment
0	1	Streaming SVE stack frame adjustment

1	0	SVE stack frame size
1	1	Streaming SVE stack frame size

SVE stack frame adjustment

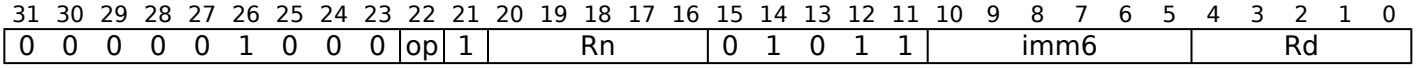
These instructions are under SVE Stack Allocation.



Decode fields op	Instruction Details
0	ADDVL
1	ADDPL

Streaming SVE stack frame adjustment

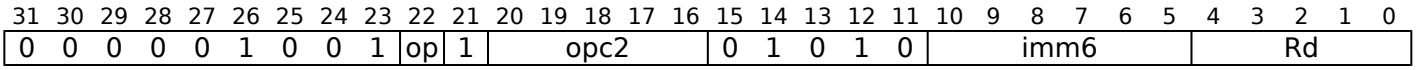
These instructions are under SVE Stack Allocation.



Decode fields op	Instruction Details	Feature
0	ADDSVL	FEAT_SME
1	ADDSPL	FEAT_SME

SVE stack frame size

These instructions are under SVE Stack Allocation.



Decode fields op	opc2	Instruction Details
0	0xxxx	UNALLOCATED
0	10xxx	UNALLOCATED
0	110xx	UNALLOCATED
0	1110x	UNALLOCATED
0	11110	UNALLOCATED
0	11111	RDVL
1		UNALLOCATED

Streaming SVE stack frame size

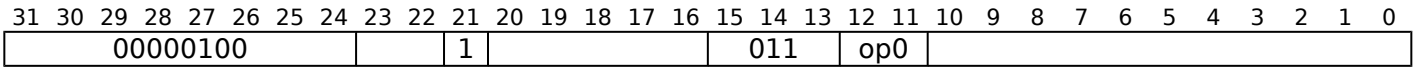
These instructions are under SVE Stack Allocation.

Decode fields op	opc2	Instruction Details	Feature
0	0xxxx	UNALLOCATED	-
0	10xxx	UNALLOCATED	-

Decode fields op	opc2	Instruction Details	Feature
0	110xx	UNALLOCATED	-
0	1110x	UNALLOCATED	-
0	11110	UNALLOCATED	-
0	11111	RDSVL	FEAT_SME
1		UNALLOCATED	-

SVE2 Integer Multiply - Unpredicated

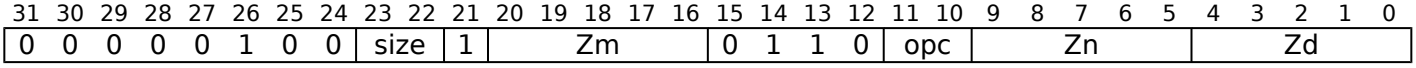
These instructions are under SVE encodings.



Decode fields op0	Instruction details
0x	SVE2 integer multiply vectors (unpredicated)
10	SVE2 signed saturating doubling multiply high (unpredicated)
11	UNALLOCATED

SVE2 integer multiply vectors (unpredicated)

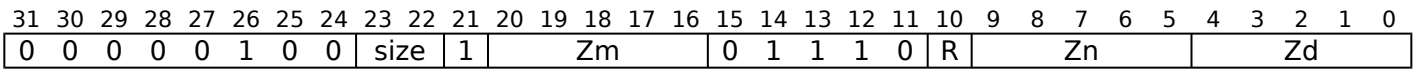
These instructions are under SVE2 Integer Multiply - Unpredicated.



Decode fields size	opc	Instruction Details
	00	MUL (vectors, unpredicated)
	10	SMULH (unpredicated)
	11	UMULH (unpredicated)
00	01	PMUL
01	01	UNALLOCATED
1x	01	UNALLOCATED

SVE2 signed saturating doubling multiply high (unpredicated)

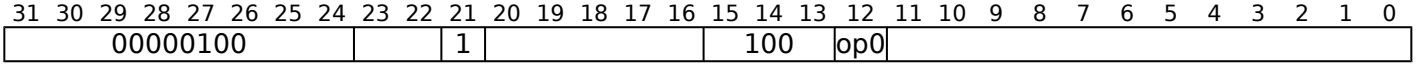
These instructions are under SVE2 Integer Multiply - Unpredicated.



Decode fields R	Instruction Details
0	SQDMULH (vectors)
1	SQRDMULH (vectors)

SVE Bitwise Shift - Unpredicated

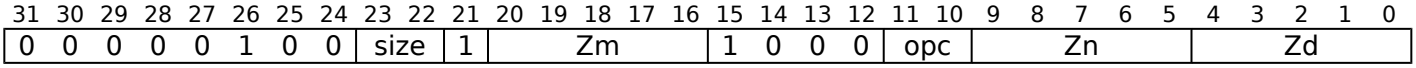
These instructions are under SVE encodings.



Decode fields op0	Instruction details
0	SVE bitwise shift by wide elements (unpredicated)
1	SVE bitwise shift by immediate (unpredicated)

SVE bitwise shift by wide elements (unpredicated)

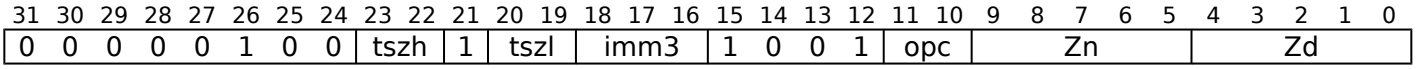
These instructions are under SVE Bitwise Shift - Unpredicated.



Decode fields opc	Instruction Details
00	ASR (wide elements, unpredicated)
01	LSR (wide elements, unpredicated)
10	UNALLOCATED
11	LSL (wide elements, unpredicated)

SVE bitwise shift by immediate (unpredicated)

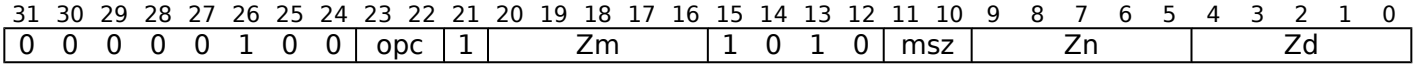
These instructions are under SVE Bitwise Shift - Unpredicated.



Decode fields opc	Instruction Details
00	ASR (immediate, unpredicated)
01	LSR (immediate, unpredicated)
10	UNALLOCATED
11	LSL (immediate, unpredicated)

SVE address generation

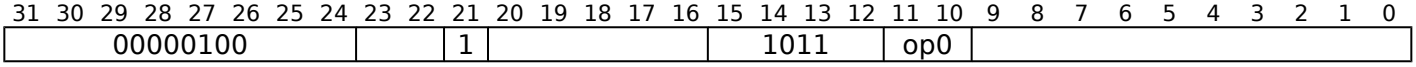
These instructions are under SVE encodings.



Decode fields opc	Instruction Details
00	ADR — Unpacked 32-bit signed offsets
01	ADR — Unpacked 32-bit unsigned offsets
1x	ADR — Packed offsets

SVE Integer Misc - Unpredicated

These instructions are under SVE encodings.

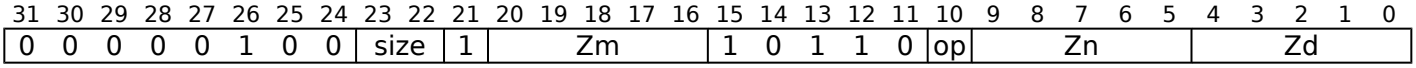


Decode fields op0	Instruction details

0x	SVE floating-point trig select coefficient
10	SVE floating-point exponential accelerator
11	SVE constructive prefix (unpredicated)

SVE floating-point trig select coefficient

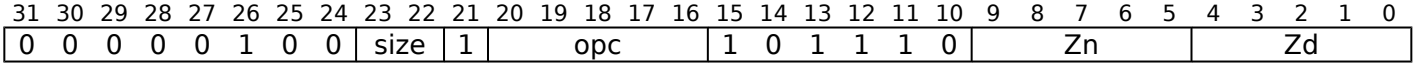
These instructions are under SVE Integer Misc - Unpredicated.



Decode fields op	Instruction Details
0	FTSSEL
1	UNALLOCATED

SVE floating-point exponential accelerator

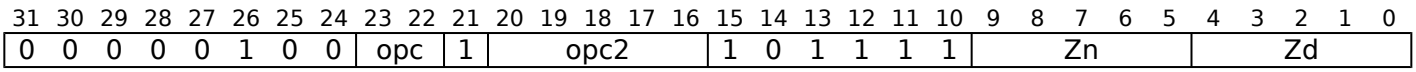
These instructions are under SVE Integer Misc - Unpredicated.



Decode fields opc	Instruction Details
00000	FEXPA
00001	UNALLOCATED
0001x	UNALLOCATED
001xx	UNALLOCATED
01xxx	UNALLOCATED
1xxxx	UNALLOCATED

SVE constructive prefix (unpredicated)

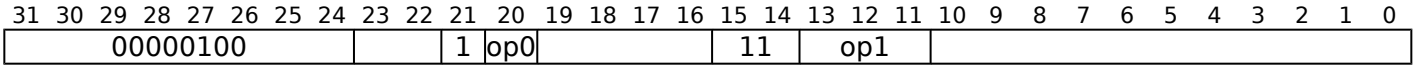
These instructions are under SVE Integer Misc - Unpredicated.



Decode fields opc	opc2	Instruction Details
00	00000	MOVPRFX (unpredicated)
00	00001	UNALLOCATED
00	0001x	UNALLOCATED
00	001xx	UNALLOCATED
00	01xxx	UNALLOCATED
00	1xxxx	UNALLOCATED
01		UNALLOCATED
1x		UNALLOCATED

SVE Element Count

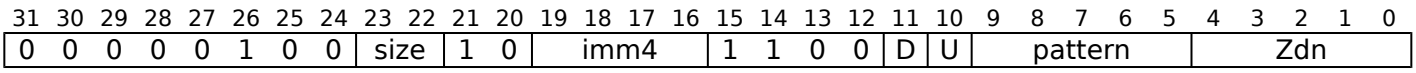
These instructions are under SVE encodings.



Decode fields op0	op1	Instruction details
0	00x	SVE saturating inc/dec vector by element count
0	100	SVE element count
0	101	UNALLOCATED
1	000	SVE inc/dec vector by element count
1	100	SVE inc/dec register by element count
1	x01	UNALLOCATED
	01x	UNALLOCATED
	11x	SVE saturating inc/dec register by element count

SVE saturating inc/dec vector by element count

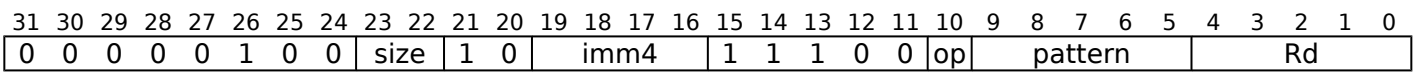
These instructions are under SVE Element Count.



Decode fields size	D	U	Instruction Details
00			UNALLOCATED
01	0	0	SQINCH (vector)
01	0	1	UQINCH (vector)
01	1	0	SQDECH (vector)
01	1	1	UQDECH (vector)
10	0	0	SQINCW (vector)
10	0	1	UQINCW (vector)
10	1	0	SQDECW (vector)
10	1	1	UQDECW (vector)
11	0	0	SQINCD (vector)
11	0	1	UQINCD (vector)
11	1	0	SQDECD (vector)
11	1	1	UQDECD (vector)

SVE element count

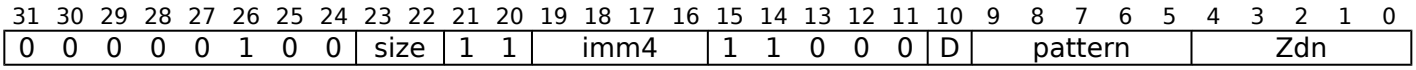
These instructions are under SVE Element Count.



Decode fields size	op	Instruction Details
	1	UNALLOCATED
00	0	CNTB, CNTD, CNTH, CNTW — CNTB
01	0	CNTB, CNTD, CNTH, CNTW — CNTH
10	0	CNTB, CNTD, CNTH, CNTW — CNTW
11	0	CNTB, CNTD, CNTH, CNTW — CNTD

SVE inc/dec vector by element count

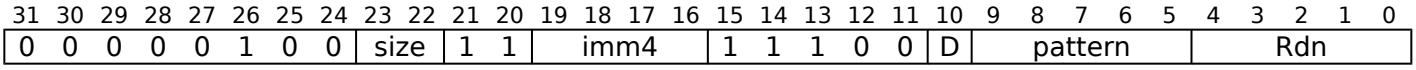
These instructions are under SVE Element Count.



Decode fields size	D	Instruction Details
00		UNALLOCATED
01	0	INCD, INCH, INCW (vector) — INCH
01	1	DECD, DECH, DECW (vector) — DECH
10	0	INCD, INCH, INCW (vector) — INCW
10	1	DECD, DECH, DECW (vector) — DECW
11	0	INCD, INCH, INCW (vector) — INCD
11	1	DECD, DECH, DECW (vector) — DECD

SVE inc/dec register by element count

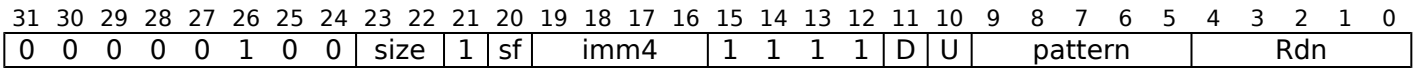
These instructions are under SVE Element Count.



Decode fields size	D	Instruction Details
00	0	INCB, INCD, INCH, INCW (scalar) — INCB
00	1	DECB, DECD, DECH, DECW (scalar) — DECB
01	0	INCB, INCD, INCH, INCW (scalar) — INCH
01	1	DECB, DECD, DECH, DECW (scalar) — DECH
10	0	INCB, INCD, INCH, INCW (scalar) — INCW
10	1	DECB, DECD, DECH, DECW (scalar) — DECW
11	0	INCB, INCD, INCH, INCW (scalar) — INCD
11	1	DECB, DECD, DECH, DECW (scalar) — DECD

SVE saturating inc/dec register by element count

These instructions are under SVE Element Count.



Decode fields size	sf	D	U	Instruction Details
00	0	0	0	SQINCB — 32-bit
00	0	0	1	UQINCB — 32-bit
00	0	1	0	SQDECB — 32-bit
00	0	1	1	UQDECB — 32-bit
00	1	0	0	SQINCB — 64-bit
00	1	0	1	UQINCB — 64-bit
00	1	1	0	SQDECB — 64-bit
00	1	1	1	UQDECB — 64-bit
01	0	0	0	SQINCH (scalar) — 32-bit
01	0	0	1	UQINCH (scalar) — 32-bit
01	0	1	0	SQDECH (scalar) — 32-bit

Decode fields size	sf	D	U	Instruction Details
01	0	1	1	UQDECH (scalar) — 32-bit
01	1	0	0	SQINCH (scalar) — 64-bit
01	1	0	1	UQINCH (scalar) — 64-bit
01	1	1	0	SQDECH (scalar) — 64-bit
01	1	1	1	UQDECH (scalar) — 64-bit
10	0	0	0	SQINCW (scalar) — 32-bit
10	0	0	1	UQINCW (scalar) — 32-bit
10	0	1	0	SQDECW (scalar) — 32-bit
10	0	1	1	UQDECW (scalar) — 32-bit
10	1	0	0	SQINCW (scalar) — 64-bit
10	1	0	1	UQINCW (scalar) — 64-bit
10	1	1	0	SQDECW (scalar) — 64-bit
10	1	1	1	UQDECW (scalar) — 64-bit
11	0	0	0	SQINCD (scalar) — 32-bit
11	0	0	1	UQINCD (scalar) — 32-bit
11	0	1	0	SQDECD (scalar) — 32-bit
11	0	1	1	UQDECD (scalar) — 32-bit
11	1	0	0	SQINCD (scalar) — 64-bit
11	1	0	1	UQINCD (scalar) — 64-bit
11	1	1	0	SQDECD (scalar) — 64-bit
11	1	1	1	UQDECD (scalar) — 64-bit

SVE Bitwise Immediate

These instructions are under SVE encodings.

31 … 25	24 … 23	22 … 21	20 … 19	18 … 0
00000101	op0	00	op1	

Decode fields op0	op1	Instruction details
11	00	DUPM
!= 11	00	SVE bitwise logical with immediate (unpredicated)
	!= 00	UNALLOCATED

SVE bitwise logical with immediate (unpredicated)

These instructions are under SVE Bitwise Immediate.

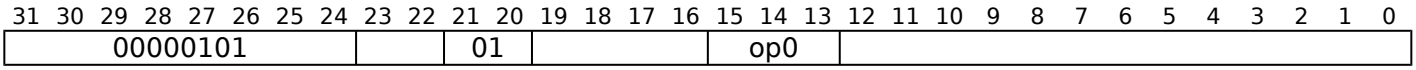
31 … 25	24 … 23	22 … 19	18 … 5	4 … 0
0 0 0 0 0 1 0 1	!= 11	0 0 0 0	imm13	Zdn
	opc			

The following constraints also apply to this encoding: opc != 11 && opc != 11

Decode fields opc	Instruction Details
00	ORR (immediate)
01	EOR (immediate)
10	AND (immediate)

SVE Integer Wide Immediate - Predicated

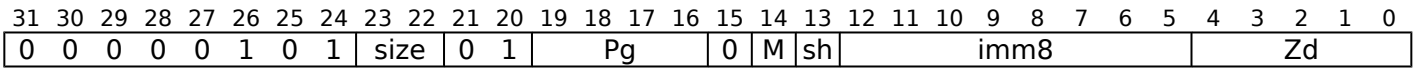
These instructions are under SVE encodings.



Decode fields op0	Instruction details
0xx	SVE copy integer immediate (predicated)
10x	UNALLOCATED
110	FCPY
111	UNALLOCATED

SVE copy integer immediate (predicated)

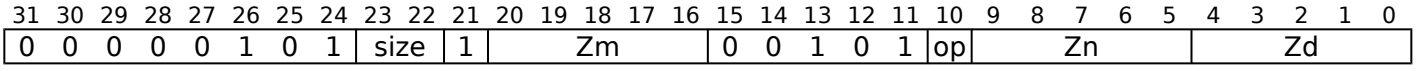
These instructions are under SVE Integer Wide Immediate - Predicated.



Decode fields M	Instruction Details
0	CPY (immediate, zeroing)
1	CPY (immediate, merging)

SVE table lookup (three sources)

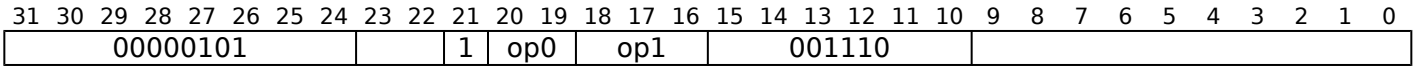
These instructions are under SVE encodings.



Decode fields op	Instruction Details
0	TBL
1	TBX

SVE Permute Vector - Unpredicated

These instructions are under SVE encodings.

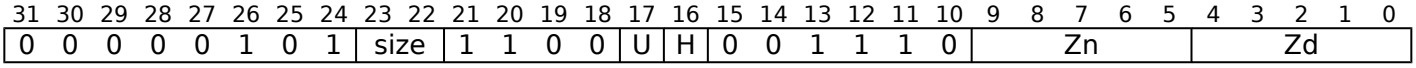


Decode fields op0	op1	Instruction details
00	000	DUP (scalar)
00	100	INSR (scalar)
00	x10	UNALLOCATED
00	xx1	UNALLOCATED
01		UNALLOCATED
10	0xx	SVE unpack vector elements
10	100	INSR (SIMD&FP scalar)
10	110	UNALLOCATED
10	1x1	UNALLOCATED

11	000	REV (vector)
11	!= 000	UNALLOCATED

SVE unpack vector elements

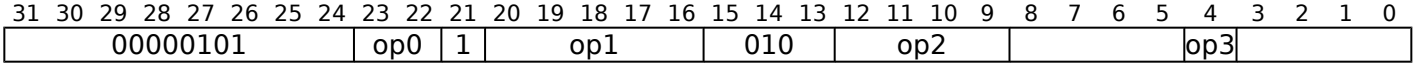
These instructions are under SVE Permute Vector - Unpredicated.



Decode fields U	H	Instruction Details
0	0	SUNPKHI, SUNPKLO — SUNPKLO
0	1	SUNPKHI, SUNPKLO — SUNPKHI
1	0	UUNPKHI, UUNPKLO — UUNPKLO
1	1	UUNPKHI, UUNPKLO — UUNPKHI

SVE Permute Predicate

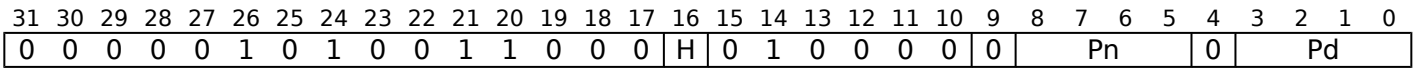
These instructions are under SVE encodings.



op0	Decode fields op1	op2	op3	Instruction details
00	1000x	0000	0	SVE unpack predicate elements
01	1000x	0000	0	UNALLOCATED
10	1000x	0000	0	UNALLOCATED
11	1000x	0000	0	UNALLOCATED
	0xxxx	xxx0	0	SVE permute predicate elements
	0xxxx	xxx1	0	UNALLOCATED
	10100	0000	0	REV (predicate)
	10101	0000	0	UNALLOCATED
	10x0x	1000	0	UNALLOCATED
	10x0x	x100	0	UNALLOCATED
	10x0x	xx10	0	UNALLOCATED
	10x0x	xxx1	0	UNALLOCATED
	10x1x		0	UNALLOCATED
	11xxx		0	UNALLOCATED
			1	UNALLOCATED

SVE unpack predicate elements

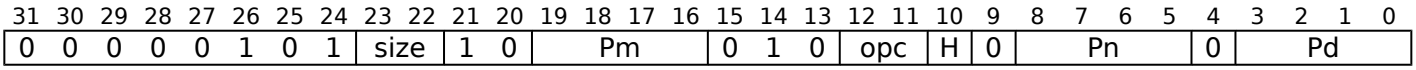
These instructions are under SVE Permute Predicate.



Decode fields H	Instruction Details
0	PUNPKHI, PUNPKLO — PUNPKLO
1	PUNPKHI, PUNPKLO — PUNPKHI

SVE permute predicate elements

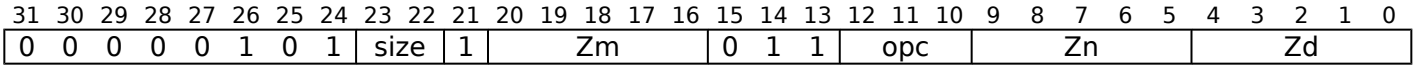
These instructions are under SVE Permute Predicate.



Decode fields opc	H	Instruction Details
00	0	ZIP1, ZIP2 (predicates) — ZIP1
00	1	ZIP1, ZIP2 (predicates) — ZIP2
01	0	UZP1, UZP2 (predicates) — UZP1
01	1	UZP1, UZP2 (predicates) — UZP2
10	0	TRN1, TRN2 (predicates) — TRN1
10	1	TRN1, TRN2 (predicates) — TRN2
11		UNALLOCATED

SVE permute vector elements

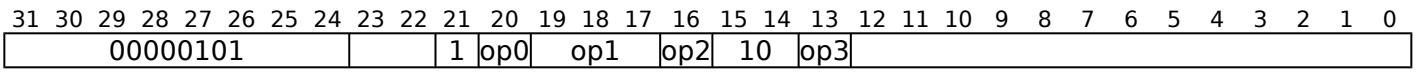
These instructions are under SVE encodings.



Decode fields opc	Instruction Details
000	ZIP1, ZIP2 (vectors) — ZIP1
001	ZIP1, ZIP2 (vectors) — ZIP2
010	UZP1, UZP2 (vectors) — UZP1
011	UZP1, UZP2 (vectors) — UZP2
100	TRN1, TRN2 (vectors) — TRN1
101	TRN1, TRN2 (vectors) — TRN2
11x	UNALLOCATED

SVE Permute Vector - Predicated

These instructions are under SVE encodings.



Decode fields op0	op1	op2	op3	Instruction details
0	000	0	0	CPY (SIMD&FP scalar)
0	000	1	0	COMPACT
0	000		1	SVE extract element to general register
0	001		0	SVE extract element to SIMD&FP scalar register
0	01x		0	SVE reverse within elements
0	01x		1	UNALLOCATED
0	100	0	1	CPY (scalar)
0	100	1	1	UNALLOCATED
0	100		0	SVE conditionally broadcast element to vector
0	101		0	SVE conditionally extract element to SIMD&FP scalar
0	110	0	0	SPLICE — Destructive

0	110	1	0	SPLICE — Constructive
0	110		1	UNALLOCATED
0	111	0	0	SVE reverse doublewords
0	111	0	1	UNALLOCATED
0	111	1		UNALLOCATED
0	x01		1	UNALLOCATED
1	000		0	UNALLOCATED
1	000		1	SVE conditionally extract element to general register
1	!= 000			UNALLOCATED

SVE extract element to general register

These instructions are under SVE Permute Vector - Predicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size		1	0	0	0	0	B	1	0	1	Pg			Zn					Rd				

Decode fields B	Instruction Details
0	LASTA (scalar)
1	LASTB (scalar)

SVE extract element to SIMD&FP scalar register

These instructions are under SVE Permute Vector - Predicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size		1	0	0	0	1	B	1	0	0	Pg			Zn					Vd				

Decode fields B	Instruction Details
0	LASTA (SIMD&FP scalar)
1	LASTB (SIMD&FP scalar)

SVE reverse within elements

These instructions are under SVE Permute Vector - Predicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size		1	0	0	1	opc		1	0	0	Pg			Zn					Zd				

Decode fields opc	Instruction Details
00	REVB, REVH, REVW — REVB
01	REVB, REVH, REVW — REVH
10	REVB, REVH, REVW — REVW
11	RBIT

SVE conditionally broadcast element to vector

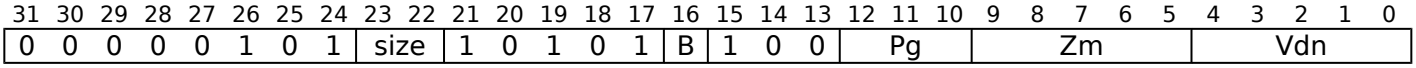
These instructions are under SVE Permute Vector - Predicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	size		1	0	1	0	0	B	1	0	0	Pg			Zm					Zdn				

Decode fields B	Instruction Details
0	CLASTA (vectors)
1	CLASTB (vectors)

SVE conditionally extract element to SIMD&FP scalar

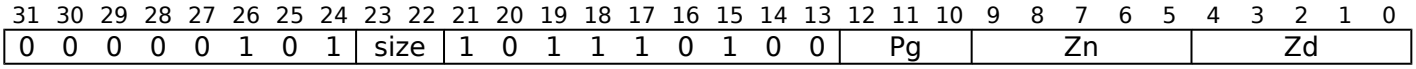
These instructions are under SVE Permute Vector - Predicated.



Decode fields B	Instruction Details
0	CLASTA (SIMD&FP scalar)
1	CLASTB (SIMD&FP scalar)

SVE reverse doublewords

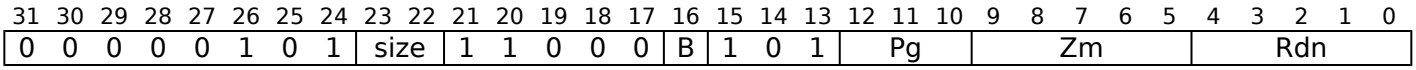
These instructions are under SVE Permute Vector - Predicated.



Decode fields size	Instruction Details	Feature
00	REVD	FEAT_SME
01	UNALLOCATED	-
1x	UNALLOCATED	-

SVE conditionally extract element to general register

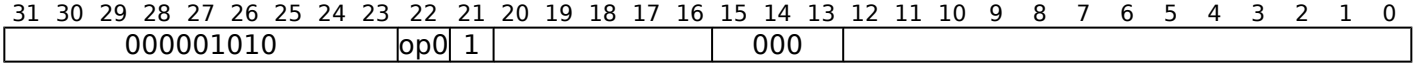
These instructions are under SVE Permute Vector - Predicated.



Decode fields B	Instruction Details
0	CLASTA (scalar)
1	CLASTB (scalar)

SVE Permute Vector - Extract

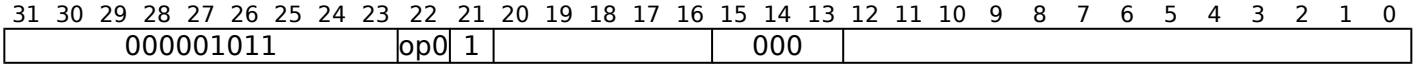
These instructions are under SVE encodings.



Decode fields op0	Instruction details
0	EXT — Destructive
1	EXT — Constructive

SVE Permute Vector - Segments

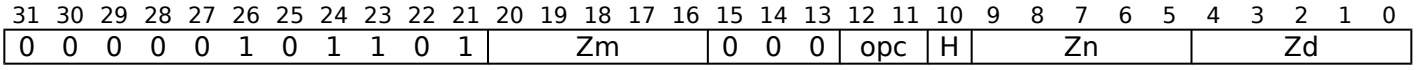
These instructions are under SVE encodings.



Decode fields op0	Instruction details
0	SVE permute vector segments
1	UNALLOCATED

SVE permute vector segments

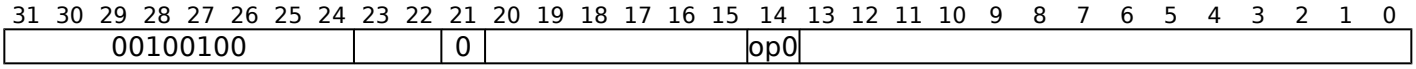
These instructions are under SVE Permute Vector - Segments.



Decode fields opc	H	Instruction Details	Feature
00	0	ZIP1, ZIP2 (vectors) — ZIP1	FEAT_F64MM
00	1	ZIP1, ZIP2 (vectors) — ZIP2	FEAT_F64MM
01	0	UZP1, UZP2 (vectors) — UZP1	FEAT_F64MM
01	1	UZP1, UZP2 (vectors) — UZP2	FEAT_F64MM
10		UNALLOCATED	-
11	0	TRN1, TRN2 (vectors) — TRN1	FEAT_F64MM
11	1	TRN1, TRN2 (vectors) — TRN2	FEAT_F64MM

SVE Integer Compare - Vectors

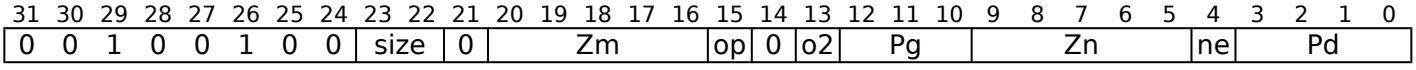
These instructions are under SVE encodings.



Decode fields op0	Instruction details
0	SVE integer compare vectors
1	SVE integer compare with wide elements

SVE integer compare vectors

These instructions are under SVE Integer Compare - Vectors.

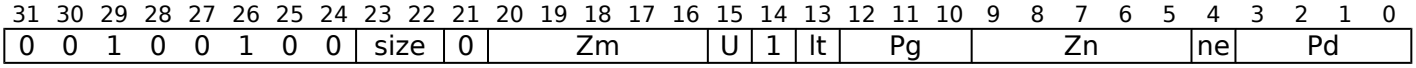


Decode fields op	o2	ne	Instruction Details
0	0	0	CMP<cc> (vectors) — CMPHS
0	0	1	CMP<cc> (vectors) — CMPHI
0	1	0	CMP<cc> (wide elements) — CMPEQ
0	1	1	CMP<cc> (wide elements) — CMPNE
1	0	0	CMP<cc> (vectors) — CMPGE
1	0	1	CMP<cc> (vectors) — CMPGT

Decode fields op	o2	ne	Instruction Details
1	1	0	CMP<cc> (vectors) — CMPEQ
1	1	1	CMP<cc> (vectors) — CMPNE

SVE integer compare with wide elements

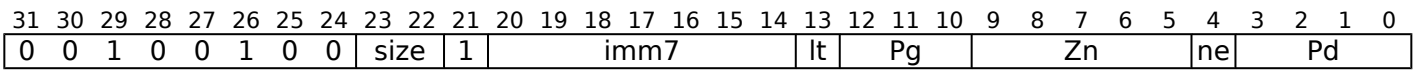
These instructions are under SVE Integer Compare - Vectors.



Decode fields U	lt	ne	Instruction Details
0	0	0	CMP<cc> (wide elements) — CMPGE
0	0	1	CMP<cc> (wide elements) — CMPGT
0	1	0	CMP<cc> (wide elements) — CMPLT
0	1	1	CMP<cc> (wide elements) — CMPLE
1	0	0	CMP<cc> (wide elements) — CMPHS
1	0	1	CMP<cc> (wide elements) — CMPHI
1	1	0	CMP<cc> (wide elements) — CMPLO
1	1	1	CMP<cc> (wide elements) — CMPLS

SVE integer compare with unsigned immediate

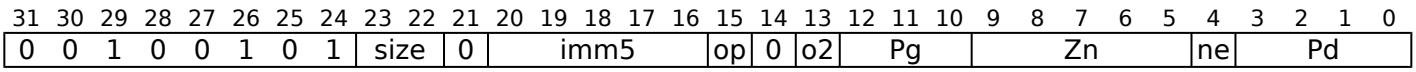
These instructions are under SVE encodings.



Decode fields lt	ne	Instruction Details
0	0	CMP<cc> (immediate) — CMPHS
0	1	CMP<cc> (immediate) — CMPHI
1	0	CMP<cc> (immediate) — CMPLO
1	1	CMP<cc> (immediate) — CMPLS

SVE integer compare with signed immediate

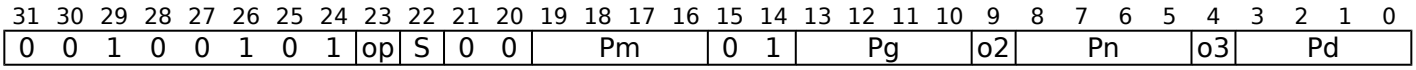
These instructions are under SVE encodings.



Decode fields op	o2	ne	Instruction Details
0	0	0	CMP<cc> (immediate) — CMPGE
0	0	1	CMP<cc> (immediate) — CMPGT
0	1	0	CMP<cc> (immediate) — CMPLT
0	1	1	CMP<cc> (immediate) — CMPLE
1	0	0	CMP<cc> (immediate) — CMPEQ
1	0	1	CMP<cc> (immediate) — CMPNE
1	1		UNALLOCATED

SVE predicate logical operations

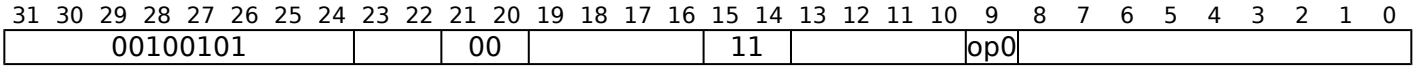
These instructions are under SVE encodings.



Decode fields op	S	o2	o3	Instruction Details
0	0	0	0	AND (predicates)
0	0	0	1	BIC (predicates)
0	0	1	0	EOR (predicates)
0	0	1	1	SEL (predicates)
0	1	0	0	ANDS
0	1	0	1	BICS
0	1	1	0	EORS
0	1	1	1	UNALLOCATED
1	0	0	0	ORR (predicates)
1	0	0	1	ORN (predicates)
1	0	1	0	NOR
1	0	1	1	NAND
1	1	0	0	ORRS
1	1	0	1	ORNS
1	1	1	0	NORS
1	1	1	1	NANDS

SVE Propagate Break

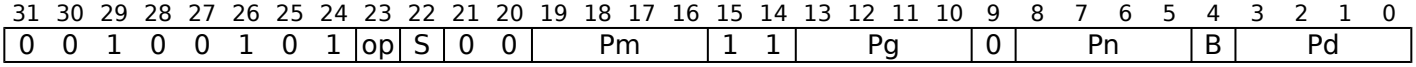
These instructions are under SVE encodings.



Decode fields op0	Instruction details
0	SVE propagate break from previous partition
1	UNALLOCATED

SVE propagate break from previous partition

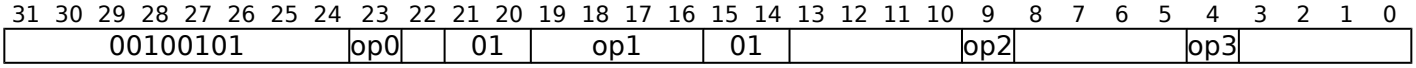
These instructions are under SVE Propagate Break.



Decode fields op	S	B	Instruction Details
0	0	0	BRKPA
0	0	1	BRKPB
0	1	0	BRKPAS
0	1	1	BRKPBS
1			UNALLOCATED

SVE Partition Break

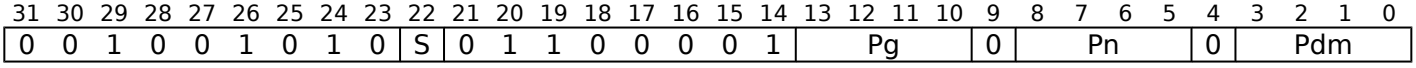
These instructions are under SVE encodings.



Decode fields op0	op1	op2	op3	Instruction details
0	1000	0	0	SVE propagate break to next partition
0	1000	0	1	UNALLOCATED
0	x000	1		UNALLOCATED
0	x1xx			UNALLOCATED
0	xx1x			UNALLOCATED
0	xxx1			UNALLOCATED
1	0000	1		UNALLOCATED
1	!= 0000			UNALLOCATED
	0000	0		SVE partition break condition

SVE propagate break to next partition

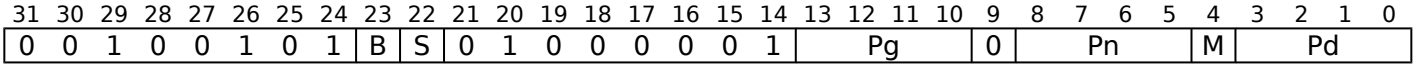
These instructions are under SVE Partition Break.



Decode fields S	Instruction Details
0	BRKN
1	BRKNS

SVE partition break condition

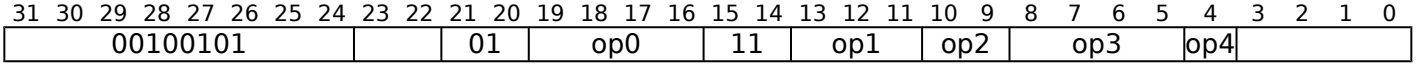
These instructions are under SVE Partition Break.



Decode fields B	S	M	Instruction Details
	1	1	UNALLOCATED
0	0		BRKA
0	1	0	BRKAS
1	0		BRKB
1	1	0	BRKBS

SVE Predicate Misc

These instructions are under SVE encodings.



Decode fields op0	op1	op2	op3	op4	Instruction details
0000		x0		0	SVE predicate test

0100		x0		0	UNALLOCATED
0x10		x0		0	UNALLOCATED
0xx1		x0		0	UNALLOCATED
0xxx		x1		0	UNALLOCATED
1000	000	00		0	SVE predicate first active
1000	000	!= 00		0	UNALLOCATED
1000	100	10	0000	0	SVE predicate zero
1000	100	10	!= 0000	0	UNALLOCATED
1000	110	00		0	SVE predicate read from FFR (predicated)
1001	000	0x		0	UNALLOCATED
1001	000	10		0	PNEXT
1001	000	11		0	UNALLOCATED
1001	100	10		0	UNALLOCATED
1001	110	00	0000	0	SVE predicate read from FFR (unpredicated)
1001	110	00	!= 0000	0	UNALLOCATED
100x	010			0	UNALLOCATED
100x	100	0x		0	SVE predicate initialize
100x	100	11		0	UNALLOCATED
100x	110	!= 00		0	UNALLOCATED
100x	xx1			0	UNALLOCATED
110x				0	UNALLOCATED
1x1x				0	UNALLOCATED
				1	UNALLOCATED

SVE predicate test

These instructions are under SVE Predicate Misc.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	op	S	0	1	0	0	0	0	1	1	Pg				0	Pn				0	opc2			

Decode fields op	S	opc2	Instruction Details
0	0		UNALLOCATED
0	1	0000	PTEST
0	1	0001	UNALLOCATED
0	1	001x	UNALLOCATED
0	1	01xx	UNALLOCATED
0	1	1xxx	UNALLOCATED
1			UNALLOCATED

SVE predicate first active

These instructions are under SVE Predicate Misc.

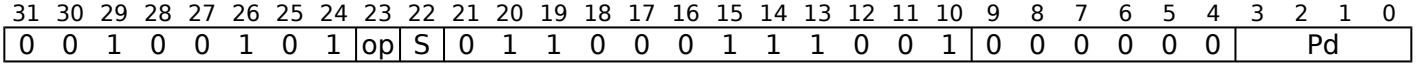
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	op	S	0	1	1	0	0	0	1	1	0	0	0	0	0	Pg				0	Pdn			

Decode fields op	S	Instruction Details
0	0	UNALLOCATED
0	1	PFIRST

Decode fields op	S	Instruction Details
1		UNALLOCATED

SVE predicate zero

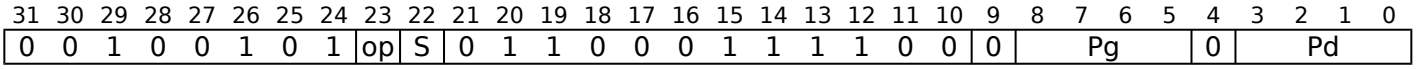
These instructions are under SVE Predicate Misc.



Decode fields op	S	Instruction Details
0	0	PFALSE
0	1	UNALLOCATED
1		UNALLOCATED

SVE predicate read from FFR (predicated)

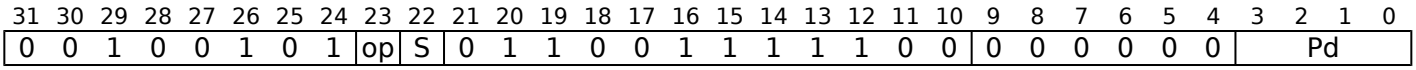
These instructions are under SVE Predicate Misc.



Decode fields op	S	Instruction Details
0	0	RDFFR (predicated)
0	1	RDFFRS
1		UNALLOCATED

SVE predicate read from FFR (unpredicated)

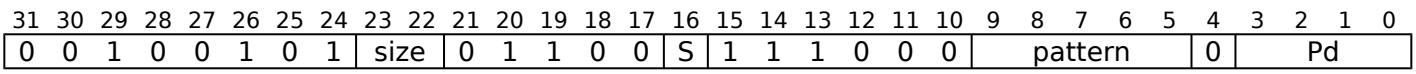
These instructions are under SVE Predicate Misc.



Decode fields op	S	Instruction Details
0	0	RDFFR (unpredicated)
0	1	UNALLOCATED
1		UNALLOCATED

SVE predicate initialize

These instructions are under SVE Predicate Misc.



Decode fields S	Instruction Details
0	PTRUE
1	PTRUES

SVE Integer Compare - Scalars

These instructions are under SVE encodings.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00100101										1						00		op0		op1								op2			

Decode fields op0	op1	op2	Instruction details
0x			SVE integer compare scalar count and limit
10	00	0000	SVE conditionally terminate scalars
10	00	!= 0000	UNALLOCATED
11	00		SVE pointer conflict compare
1x	!= 00		UNALLOCATED

SVE integer compare scalar count and limit

These instructions are under SVE Integer Compare - Scalars.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size		1		Rm				0	0	0	sf	U	lt		Rn				eq		Pd		

Decode fields U	lt	eq	Instruction Details
0	0	0	WHILEGE
0	0	1	WHILEGT
0	1	0	WHILELT
0	1	1	WHILELE
1	0	0	WHILEHS
1	0	1	WHILEHI
1	1	0	WHILELO
1	1	1	WHILELS

SVE conditionally terminate scalars

These instructions are under SVE Integer Compare - Scalars.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	op	sz	1		Rm				0	0	1	0	0	0		Rn				ne	0	0	0	0

Decode fields op	ne	Instruction Details
0		UNALLOCATED
1	0	CTERMEQ, CTERMNE — CTERMEQ
1	1	CTERMEQ, CTERMNE — CTERMNE

SVE pointer conflict compare

These instructions are under SVE Integer Compare - Scalars.

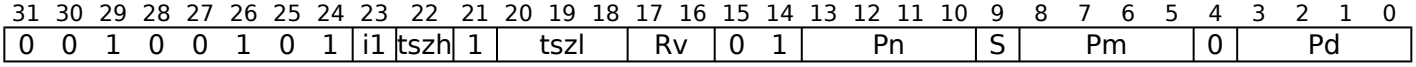
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	size		1		Rm				0	0	1	1	0	0		Rn				rw		Pd		

Decode fields rw	Instruction Details
0	WHILEWR

Decode fields rw	Instruction Details
1	WHILERW

SVE broadcast predicate element

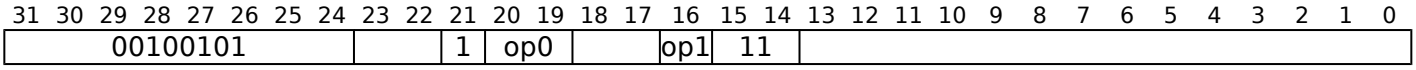
These instructions are under SVE encodings.



Decode fields S	Instruction Details	Feature
0	PSEL	FEAT_SME
1	UNALLOCATED	-

SVE Integer Wide Immediate - Unpredicated

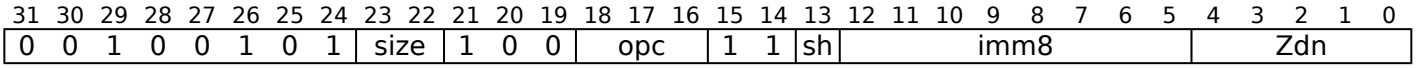
These instructions are under SVE encodings.



Decode fields op0	op1	Instruction details
00		SVE integer add/subtract immediate (unpredicated)
01		SVE integer min/max immediate (unpredicated)
10		SVE integer multiply immediate (unpredicated)
11	0	SVE broadcast integer immediate (unpredicated)
11	1	SVE broadcast floating-point immediate (unpredicated)

SVE integer add/subtract immediate (unpredicated)

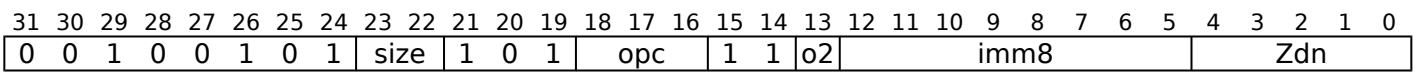
These instructions are under SVE Integer Wide Immediate - Unpredicated.



Decode fields opc	Instruction Details
000	ADD (immediate)
001	SUB (immediate)
010	UNALLOCATED
011	SUBR (immediate)
100	SQADD (immediate)
101	UQADD (immediate)
110	SQSUB (immediate)
111	UQSUB (immediate)

SVE integer min/max immediate (unpredicated)

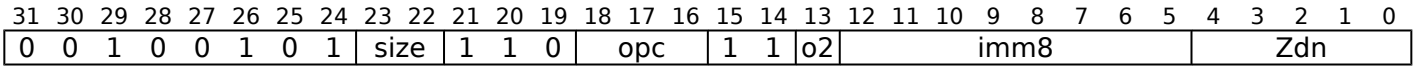
These instructions are under SVE Integer Wide Immediate - Unpredicated.



Decode fields opc	o2	Instruction Details
0xx	1	UNALLOCATED
000	0	SMAX (immediate)
001	0	UMAX (immediate)
010	0	SMIN (immediate)
011	0	UMIN (immediate)
1xx		UNALLOCATED

SVE integer multiply immediate (unpredicated)

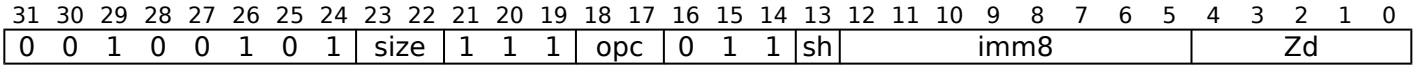
These instructions are under SVE Integer Wide Immediate - Unpredicated.



Decode fields opc	o2	Instruction Details
000	0	MUL (immediate)
000	1	UNALLOCATED
001		UNALLOCATED
01x		UNALLOCATED
1xx		UNALLOCATED

SVE broadcast integer immediate (unpredicated)

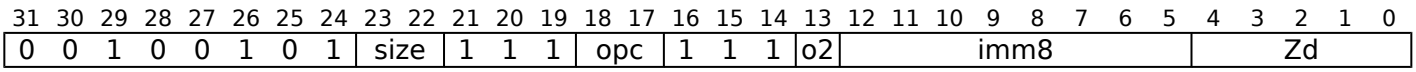
These instructions are under SVE Integer Wide Immediate - Unpredicated.



Decode fields opc	Instruction Details
00	DUP (immediate)
01	UNALLOCATED
1x	UNALLOCATED

SVE broadcast floating-point immediate (unpredicated)

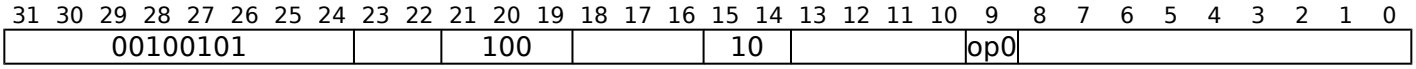
These instructions are under SVE Integer Wide Immediate - Unpredicated.



Decode fields opc	o2	Instruction Details
00	0	FDUP
00	1	UNALLOCATED
01		UNALLOCATED
1x		UNALLOCATED

SVE Predicate Count

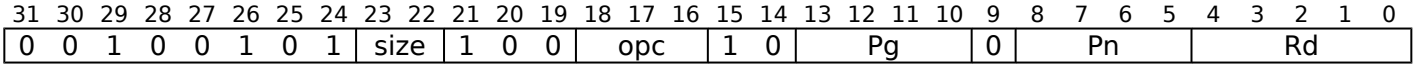
These instructions are under SVE encodings.



Decode fields op0	Instruction details
0	SVE predicate count
1	UNALLOCATED

SVE predicate count

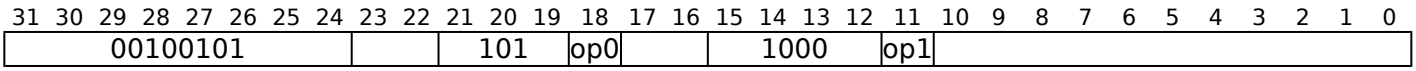
These instructions are under SVE Predicate Count.



Decode fields opc	Instruction Details
000	CNTP
001	UNALLOCATED
01x	UNALLOCATED
1xx	UNALLOCATED

SVE Inc/Dec by Predicate Count

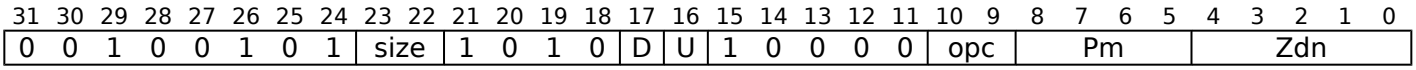
These instructions are under SVE encodings.



Decode fields op0	op1	Instruction details
0	0	SVE saturating inc/dec vector by predicate count
0	1	SVE saturating inc/dec register by predicate count
1	0	SVE inc/dec vector by predicate count
1	1	SVE inc/dec register by predicate count

SVE saturating inc/dec vector by predicate count

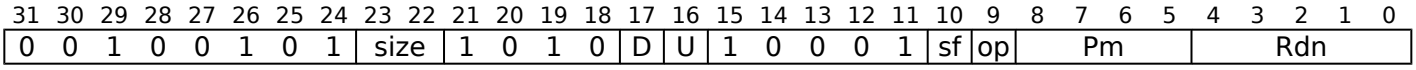
These instructions are under SVE Inc/Dec by Predicate Count.



Decode fields D	U	opc	Instruction Details
		01	UNALLOCATED
		1x	UNALLOCATED
0	0	00	SQINCP (vector)
0	1	00	UQINCP (vector)
1	0	00	SQDECP (vector)
1	1	00	UQDECP (vector)

SVE saturating inc/dec register by predicate count

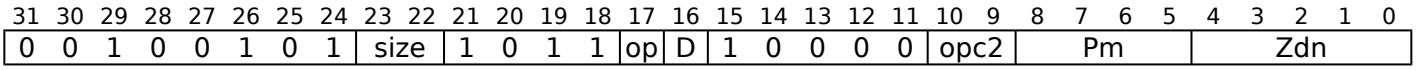
These instructions are under SVE Inc/Dec by Predicate Count.



Decode fields D	U	sf	op	Instruction Details
			1	UNALLOCATED
0	0	0	0	SQINCP (scalar) — 32-bit
0	0	1	0	SQINCP (scalar) — 64-bit
0	1	0	0	UQINCP (scalar) — 32-bit
0	1	1	0	UQINCP (scalar) — 64-bit
1	0	0	0	SQDECP (scalar) — 32-bit
1	0	1	0	SQDECP (scalar) — 64-bit
1	1	0	0	UQDECP (scalar) — 32-bit
1	1	1	0	UQDECP (scalar) — 64-bit

SVE inc/dec vector by predicate count

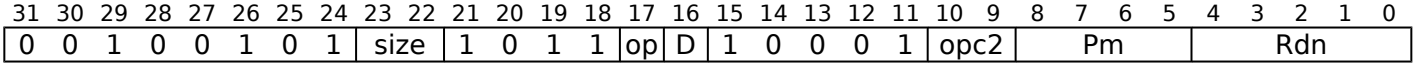
These instructions are under SVE Inc/Dec by Predicate Count.



Decode fields op	D	opc2	Instruction Details
0		01	UNALLOCATED
0		1x	UNALLOCATED
0	0	00	INCP (vector)
0	1	00	DECP (vector)
1			UNALLOCATED

SVE inc/dec register by predicate count

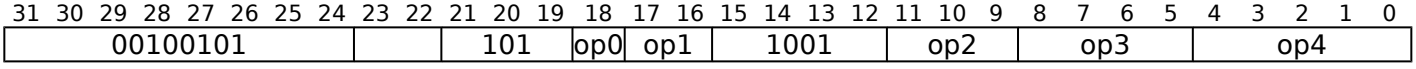
These instructions are under SVE Inc/Dec by Predicate Count.



Decode fields op	D	opc2	Instruction Details
0		01	UNALLOCATED
0		1x	UNALLOCATED
0	0	00	INCP (scalar)
0	1	00	DECP (scalar)
1			UNALLOCATED

SVE Write FFR

These instructions are under SVE encodings.



Decode fields op0	op1	op2	op3	op4	Instruction details
0	00	000		00000	SVE FFR write from predicate

1	00	000	0000	00000	SVE FFR initialise
1	00	000	1xxx	00000	UNALLOCATED
1	00	000	x1xx	00000	UNALLOCATED
1	00	000	xx1x	00000	UNALLOCATED
1	00	000	xxx1	00000	UNALLOCATED
	00	000		!= 00000	UNALLOCATED
	00	!= 000			UNALLOCATED
	!= 00				UNALLOCATED

SVE FFR write from predicate

These instructions are under SVE Write FFR.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	opc		1	0	1	0	0	0	1	0	0	1	0	0	0	Pn				0	0	0	0	0

Decode fields opc	Instruction Details
00	WRFFR
01	UNALLOCATED
1x	UNALLOCATED

SVE FFR initialise

These instructions are under SVE Write FFR.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	opc		1	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0

Decode fields opc	Instruction Details
00	SETFFR
01	UNALLOCATED
1x	UNALLOCATED

SVE Integer Multiply-Add - Unpredicated

These instructions are under SVE encodings.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01000100										0						0	op0														

Decode fields op0	Instruction details
0000x	SVE integer dot product (unpredicated)
0001x	SVE2 saturating multiply-add interleaved long
001xx	CDOT (vectors)
01xxx	SVE2 complex integer multiply-add
10xxx	SVE2 integer multiply-add long
110xx	SVE2 saturating multiply-add long
1110x	SVE2 saturating multiply-add high
11110	SVE mixed sign dot product
11111	UNALLOCATED

SVE integer dot product (unpredicated)

These instructions are under SVE Integer Multiply-Add - Unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size		0	Zm					0	0	0	0	0	U	Zn					Zda				

Decode fields U	Instruction Details
0	SDOT (vectors)
1	UDOT (vectors)

SVE2 saturating multiply-add interleaved long

These instructions are under SVE Integer Multiply-Add - Unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size		0	Zm					0	0	0	0	1	S	Zn					Zda				

Decode fields S	Instruction Details
0	SQDMLALBT
1	SQDMLSLBT

SVE2 complex integer multiply-add

These instructions are under SVE Integer Multiply-Add - Unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size		0	Zm					0	0	1	op	rot		Zn					Zda				

Decode fields op	Instruction Details
0	CMLA (vectors)
1	SQRDCMLAH (vectors)

SVE2 integer multiply-add long

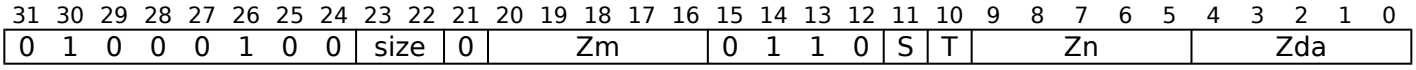
These instructions are under SVE Integer Multiply-Add - Unpredicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size		0	Zm					0	1	0	S	U	T	Zn					Zda				

Decode fields S	U	T	Instruction Details
0	0	0	SMLALB (vectors)
0	0	1	SMLALT (vectors)
0	1	0	UMLALB (vectors)
0	1	1	UMLALT (vectors)
1	0	0	SMLSLB (vectors)
1	0	1	SMLSLT (vectors)
1	1	0	UMLSLB (vectors)
1	1	1	UMLSLT (vectors)

SVE2 saturating multiply-add long

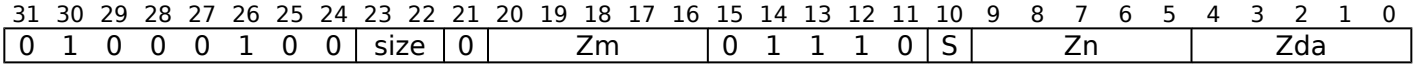
These instructions are under SVE Integer Multiply-Add - Unpredicated.



Decode fields S	T	Instruction Details
0	0	SQDMLALB (vectors)
0	1	SQDMLALT (vectors)
1	0	SQDMLSLB (vectors)
1	1	SQDMLSLT (vectors)

SVE2 saturating multiply-add high

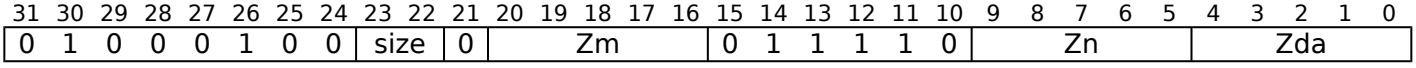
These instructions are under SVE Integer Multiply-Add - Unpredicated.



Decode fields S	Instruction Details
0	SQRDMLAH (vectors)
1	SQRDMLSH (vectors)

SVE mixed sign dot product

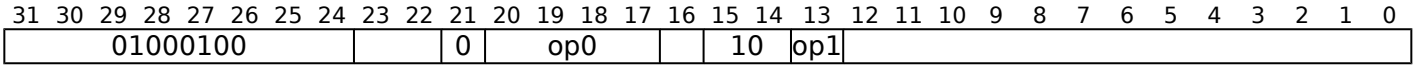
These instructions are under SVE Integer Multiply-Add - Unpredicated.



Decode fields size	Instruction Details	Feature
0x	UNALLOCATED	-
10	USDOT (vectors)	FEAT_I8MM
11	UNALLOCATED	-

SVE2 Integer - Predicated

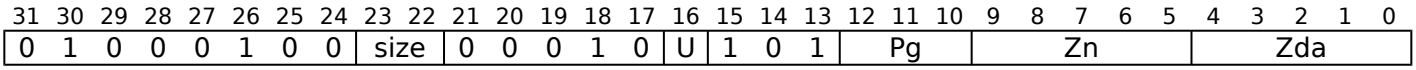
These instructions are under SVE encodings.



Decode fields op0	op1	Instruction details
0010	1	SVE2 integer pairwise add and accumulate long
0011	1	UNALLOCATED
011x	1	UNALLOCATED
0x0x	1	SVE2 integer unary operations (predicated)
0xxx	0	SVE2 saturating/rounding bitwise shift left (predicated)
10xx	0	SVE2 integer halving add/subtract (predicated)
10xx	1	SVE2 integer pairwise arithmetic
11xx	0	SVE2 saturating add/subtract
11xx	1	UNALLOCATED

SVE2 integer pairwise add and accumulate long

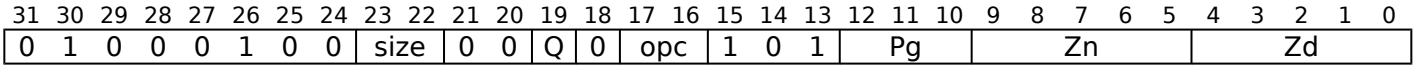
These instructions are under SVE2 Integer - Predicated.



Decode fields U	Instruction Details
0	SADALP
1	UADALP

SVE2 integer unary operations (predicated)

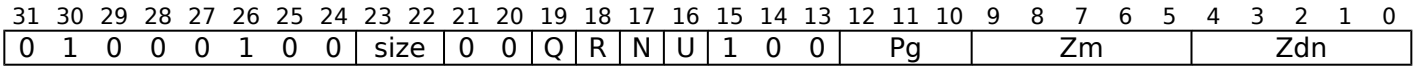
These instructions are under SVE2 Integer - Predicated.



Decode fields Q	opc	Instruction Details
	1x	UNALLOCATED
0	00	URECPE
0	01	URSQRTE
1	00	SQABS
1	01	SQNEG

SVE2 saturating/rounding bitwise shift left (predicated)

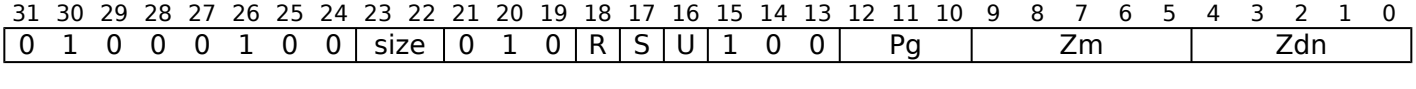
These instructions are under SVE2 Integer - Predicated.



Decode fields Q	R	N	U	Instruction Details
0		0		UNALLOCATED
0	0	1	0	SRSHL
0	0	1	1	URSHL
0	1	1	0	SRSHLR
0	1	1	1	URSHLR
1	0	0	0	SQSHL (vectors)
1	0	0	1	UQSHL (vectors)
1	0	1	0	SQRSHL
1	0	1	1	UQRSHL
1	1	0	0	SQSHLR
1	1	0	1	UQSHLR
1	1	1	0	SQRSHLR
1	1	1	1	UQRSHLR

SVE2 integer halving add/subtract (predicated)

These instructions are under SVE2 Integer - Predicated.



Decode fields R	S	U	Instruction Details
0	0	0	SHADD
0	0	1	UHADD
0	1	0	SHSUB
0	1	1	UHSUB
1	0	0	SRHADD
1	0	1	URHADD
1	1	0	SHSUBR
1	1	1	UHSUBR

SVE2 integer pairwise arithmetic

These instructions are under SVE2 Integer - Predicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size		0	1	0	opc		U	1	0	1	Pg			Zm					Zdn				

Decode fields opc	U	Instruction Details
00	0	UNALLOCATED
00	1	ADDP
01		UNALLOCATED
10	0	SMAXP
10	1	UMAXP
11	0	SMINP
11	1	UMINP

SVE2 saturating add/subtract

These instructions are under SVE2 Integer - Predicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size		0	1	1	op	S	U	1	0	0	Pg			Zm					Zdn				

Decode fields op	S	U	Instruction Details
0	0	0	SQADD (vectors, predicated)
0	0	1	UQADD (vectors, predicated)
0	1	0	SQSUB (vectors, predicated)
0	1	1	UQSUB (vectors, predicated)
1	0	0	SUQADD
1	0	1	USQADD
1	1	0	SQSUBR
1	1	1	UQSUBR

SVE integer clamp

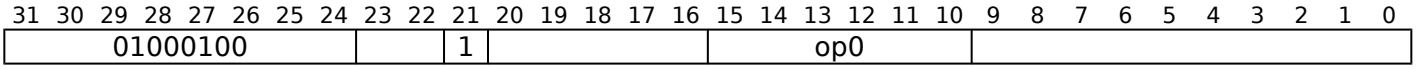
These instructions are under SVE encodings.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size		0	Zm					1	1	0	0	0	U	Zn					Zd				

Decode fields U	Instruction Details	Feature
0	SCLAMP	FEAT_SME
1	UCLAMP	FEAT_SME

SVE Multiply - Indexed

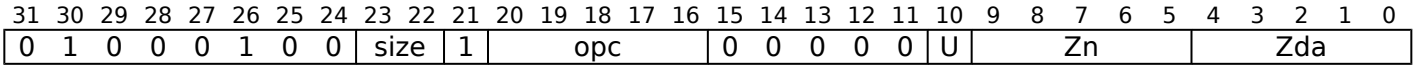
These instructions are under SVE encodings.



Decode fields op0	Instruction details
00000x	SVE integer dot product (indexed)
00001x	SVE2 integer multiply-add (indexed)
00010x	SVE2 saturating multiply-add high (indexed)
00011x	SVE mixed sign dot product (indexed)
001xxx	SVE2 saturating multiply-add (indexed)
0100xx	SVE2 complex integer dot product (indexed)
0101xx	UNALLOCATED
0110xx	SVE2 complex integer multiply-add (indexed)
0111xx	SVE2 complex saturating multiply-add (indexed)
10xxxx	SVE2 integer multiply-add long (indexed)
110xxx	SVE2 integer multiply long (indexed)
1110xx	SVE2 saturating multiply (indexed)
11110x	SVE2 saturating multiply high (indexed)
111110	SVE2 integer multiply (indexed)
111111	UNALLOCATED

SVE integer dot product (indexed)

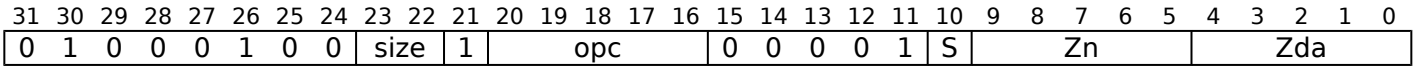
These instructions are under SVE Multiply - Indexed.



Decode fields size	Decode fields U	Instruction Details
0x		UNALLOCATED
10	0	SDOT (indexed) — 32-bit
10	1	UDOT (indexed) — 32-bit
11	0	SDOT (indexed) — 64-bit
11	1	UDOT (indexed) — 64-bit

SVE2 integer multiply-add (indexed)

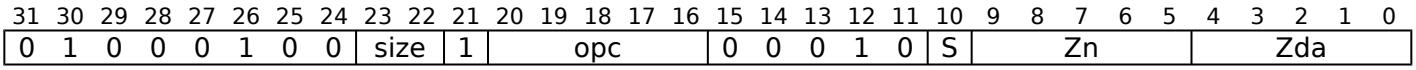
These instructions are under SVE Multiply - Indexed.



Decode fields size	S	Instruction Details
0x	0	MLA (indexed) — 16-bit
0x	1	MLS (indexed) — 16-bit
10	0	MLA (indexed) — 32-bit
10	1	MLS (indexed) — 32-bit
11	0	MLA (indexed) — 64-bit
11	1	MLS (indexed) — 64-bit

SVE2 saturating multiply-add high (indexed)

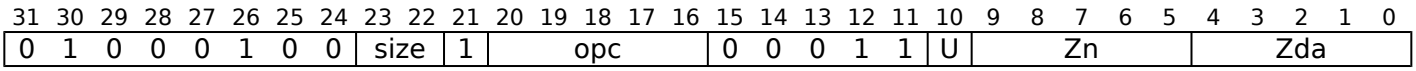
These instructions are under SVE Multiply - Indexed.



Decode fields size	S	Instruction Details
0x	0	SQRDMLAH (indexed) — 16-bit
0x	1	SQRDMLSH (indexed) — 16-bit
10	0	SQRDMLAH (indexed) — 32-bit
10	1	SQRDMLSH (indexed) — 32-bit
11	0	SQRDMLAH (indexed) — 64-bit
11	1	SQRDMLSH (indexed) — 64-bit

SVE mixed sign dot product (indexed)

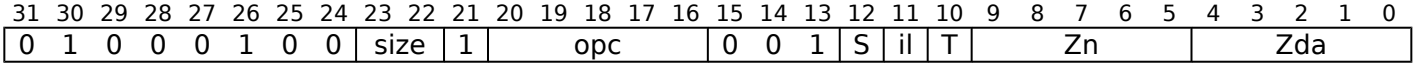
These instructions are under SVE Multiply - Indexed.



Decode fields size	U	Instruction Details	Feature
0x		UNALLOCATED	-
10	0	USDOT (indexed)	FEAT_I8MM
10	1	SUDOT	FEAT_I8MM
11		UNALLOCATED	-

SVE2 saturating multiply-add (indexed)

These instructions are under SVE Multiply - Indexed.



Decode fields size	S	T	Instruction Details
0x			UNALLOCATED
10	0	0	SQDMLALB (indexed) — 32-bit
10	0	1	SQDMLALT (indexed) — 32-bit
10	1	0	SQDMLSLB (indexed) — 32-bit
10	1	1	SQDMLSLT (indexed) — 32-bit
11	0	0	SQDMLALB (indexed) — 64-bit

Decode fields size	S	T	Instruction Details
11	0	1	SQDMLALT (indexed) — 64-bit
11	1	0	SQDMLSLB (indexed) — 64-bit
11	1	1	SQDMLSLT (indexed) — 64-bit

SVE2 complex integer dot product (indexed)

These instructions are under SVE Multiply - Indexed.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size		1	opc					0	1	0	0	rot		Zn					Zda				

Decode fields size	Instruction Details
0x	UNALLOCATED
10	CDOT (indexed) — 32-bit
11	CDOT (indexed) — 64-bit

SVE2 complex integer multiply-add (indexed)

These instructions are under SVE Multiply - Indexed.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size		1	opc					0	1	1	0	rot		Zn					Zda				

Decode fields size	Instruction Details
0x	UNALLOCATED
10	CMLA (indexed) — 16-bit
11	CMLA (indexed) — 32-bit

SVE2 complex saturating multiply-add (indexed)

These instructions are under SVE Multiply - Indexed.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size		1	opc					0	1	1	1	rot		Zn					Zda				

Decode fields size	Instruction Details
0x	UNALLOCATED
10	SQRDCMLAH (indexed) — 16-bit
11	SQRDCMLAH (indexed) — 32-bit

SVE2 integer multiply-add long (indexed)

These instructions are under SVE Multiply - Indexed.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size		1	opc					1	0	S	U	il	T	Zn					Zda				

Decode fields size	S	U	T	Instruction Details
0x				UNALLOCATED
10	0	0	0	SMLALB (indexed) — 32-bit

Decode fields size	S	U	T	Instruction Details
10	0	0	1	SMLALT (indexed) — 32-bit
10	0	1	0	UMLALB (indexed) — 32-bit
10	0	1	1	UMLALT (indexed) — 32-bit
10	1	0	0	SMLSLB (indexed) — 32-bit
10	1	0	1	SMLSLT (indexed) — 32-bit
10	1	1	0	UMLSLB (indexed) — 32-bit
10	1	1	1	UMLSLT (indexed) — 32-bit
11	0	0	0	SMLALB (indexed) — 64-bit
11	0	0	1	SMLALT (indexed) — 64-bit
11	0	1	0	UMLALB (indexed) — 64-bit
11	0	1	1	UMLALT (indexed) — 64-bit
11	1	0	0	SMLSLB (indexed) — 64-bit
11	1	0	1	SMLSLT (indexed) — 64-bit
11	1	1	0	UMLSLB (indexed) — 64-bit
11	1	1	1	UMLSLT (indexed) — 64-bit

SVE2 integer multiply long (indexed)

These instructions are under SVE Multiply - Indexed.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size		1	opc					1	1	0	U	il	T	Zn					Zd				

Decode fields size	U	T	Instruction Details
0x			UNALLOCATED
10	0	0	SMULLB (indexed) — 32-bit
10	0	1	SMULLT (indexed) — 32-bit
10	1	0	UMULLB (indexed) — 32-bit
10	1	1	UMULLT (indexed) — 32-bit
11	0	0	SMULLB (indexed) — 64-bit
11	0	1	SMULLT (indexed) — 64-bit
11	1	0	UMULLB (indexed) — 64-bit
11	1	1	UMULLT (indexed) — 64-bit

SVE2 saturating multiply (indexed)

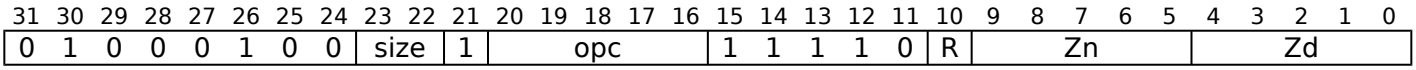
These instructions are under SVE Multiply - Indexed.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	size		1	opc					1	1	1	0	il	T	Zn					Zd				

Decode fields size	T	Instruction Details
0x		UNALLOCATED
10	0	SQDMULLB (indexed) — 32-bit
10	1	SQDMULLT (indexed) — 32-bit
11	0	SQDMULLB (indexed) — 64-bit
11	1	SQDMULLT (indexed) — 64-bit

SVE2 saturating multiply high (indexed)

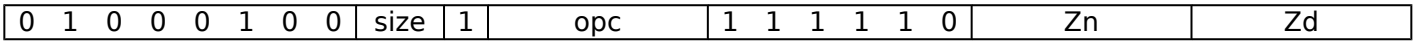
These instructions are under SVE Multiply - Indexed.



Decode fields size	R	Instruction Details
0x	0	SQDMULH (indexed) — 16-bit
0x	1	SQRDMULH (indexed) — 16-bit
10	0	SQDMULH (indexed) — 32-bit
10	1	SQRDMULH (indexed) — 32-bit
11	0	SQDMULH (indexed) — 64-bit
11	1	SQRDMULH (indexed) — 64-bit

SVE2 integer multiply (indexed)

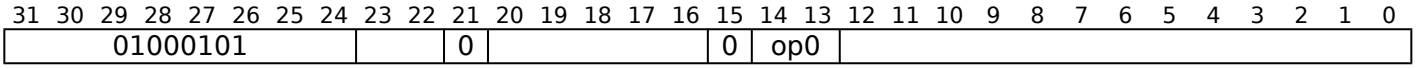
These instructions are under SVE Multiply - Indexed.



Decode fields size	Instruction Details
0x	MUL (indexed) — 16-bit
10	MUL (indexed) — 32-bit
11	MUL (indexed) — 64-bit

SVE2 Widening Integer Arithmetic

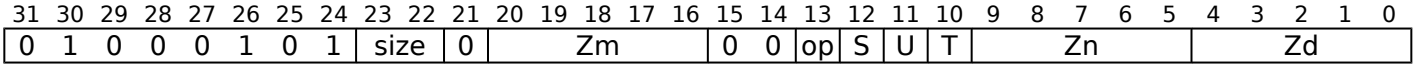
These instructions are under SVE encodings.



Decode fields op0	Instruction details
0x	SVE2 integer add/subtract long
10	SVE2 integer add/subtract wide
11	SVE2 integer multiply long

SVE2 integer add/subtract long

These instructions are under SVE2 Widening Integer Arithmetic.

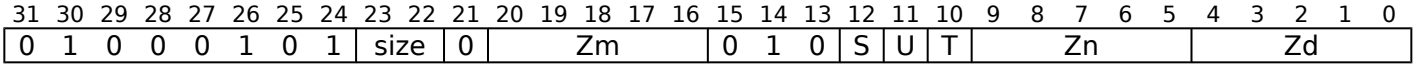


Decode fields op	S	U	T	Instruction Details
0	0	0	0	SADDLB
0	0	0	1	SADDLT
0	0	1	0	UADDLB
0	0	1	1	UADDLT
0	1	0	0	SSUBLB

Decode fields op	S	U	T	Instruction Details
0	1	0	1	SSUBLT
0	1	1	0	USUBLB
0	1	1	1	USUBLT
1	0			UNALLOCATED
1	1	0	0	SABDLB
1	1	0	1	SABDLT
1	1	1	0	UABDLB
1	1	1	1	UABDLT

SVE2 integer add/subtract wide

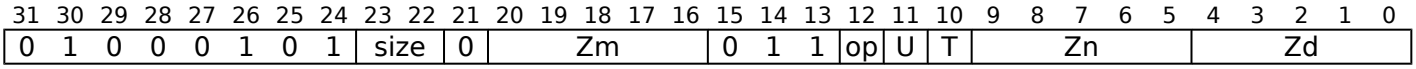
These instructions are under SVE2 Widening Integer Arithmetic.



Decode fields S	U	T	Instruction Details
0	0	0	SADDWB
0	0	1	SADDWT
0	1	0	UADDWB
0	1	1	UADDWT
1	0	0	SSUBWB
1	0	1	SSUBWT
1	1	0	USUBWB
1	1	1	USUBWT

SVE2 integer multiply long

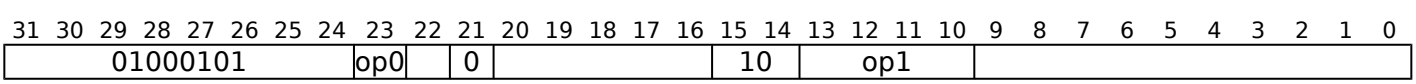
These instructions are under SVE2 Widening Integer Arithmetic.



Decode fields op	U	T	Instruction Details
0	0	0	SQDMULLB (vectors)
0	0	1	SQDMULLT (vectors)
0	1	0	PMULLB
0	1	1	PMULLT
1	0	0	SMULLB (vectors)
1	0	1	SMULLT (vectors)
1	1	0	UMULLB (vectors)
1	1	1	UMULLT (vectors)

SVE Misc

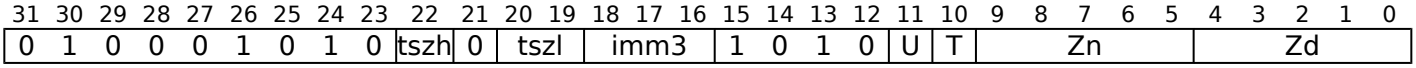
These instructions are under SVE encodings.



Decode fields op0	op1	Instruction details
0	10xx	SVE2 bitwise shift left long
1	10xx	UNALLOCATED
	00xx	SVE2 integer add/subtract interleaved long
	010x	SVE2 bitwise exclusive-or interleaved
	0110	SVE integer matrix multiply accumulate
	0111	UNALLOCATED
	11xx	SVE2 bitwise permute

SVE2 bitwise shift left long

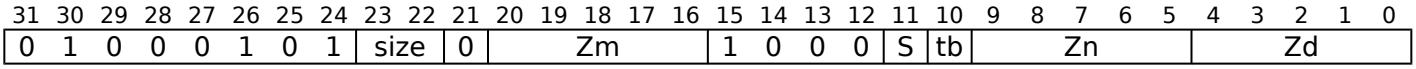
These instructions are under SVE Misc.



Decode fields U	T	Instruction Details
0	0	SSHLLB
0	1	SSHLLT
1	0	USHLLB
1	1	USHLLT

SVE2 integer add/subtract interleaved long

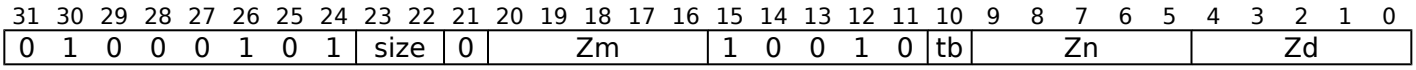
These instructions are under SVE Misc.



Decode fields S	tb	Instruction Details
0	0	SADDLBT
0	1	UNALLOCATED
1	0	SSUBLBT
1	1	SSUBLTB

SVE2 bitwise exclusive-or interleaved

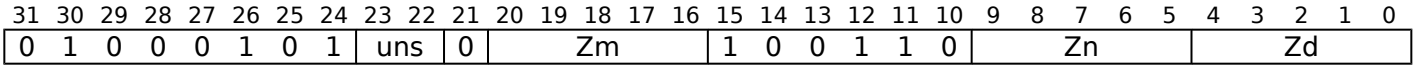
These instructions are under SVE Misc.



Decode fields tb	Instruction Details
0	EORBT
1	EORTB

SVE integer matrix multiply accumulate

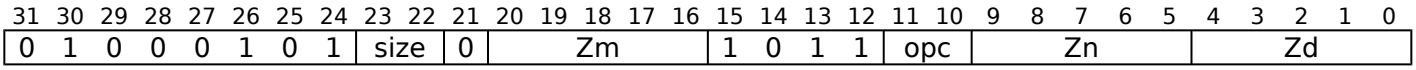
These instructions are under SVE Misc.



Decode fields uns	Instruction Details	Feature
00	SMMLA	FEAT_I8MM
01	UNALLOCATED	-
10	USMMLA	FEAT_I8MM
11	UMMLA	FEAT_I8MM

SVE2 bitwise permute

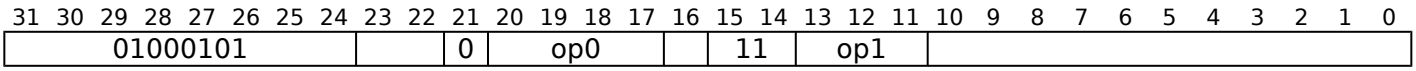
These instructions are under SVE Misc.



Decode fields opc	Instruction Details	Feature
00	BEXT	FEAT_SVE_BitPerm
01	BDEP	FEAT_SVE_BitPerm
10	BGRP	FEAT_SVE_BitPerm
11	UNALLOCATED	-

SVE2 Accumulate

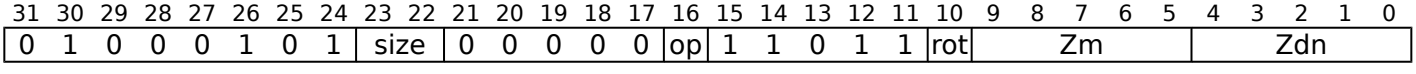
These instructions are under SVE encodings.



Decode fields op0	op1	Instruction details
0000	011	SVE2 complex integer add
!= 0000	011	UNALLOCATED
	00x	SVE2 integer absolute difference and accumulate long
	010	SVE2 integer add/subtract long with carry
	10x	SVE2 bitwise shift right and accumulate
	110	SVE2 bitwise shift and insert
	111	SVE2 integer absolute difference and accumulate

SVE2 complex integer add

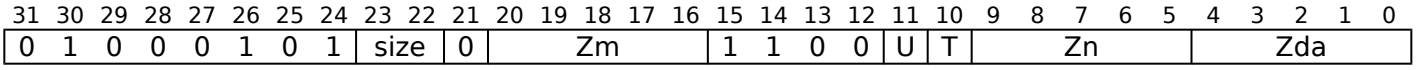
These instructions are under SVE2 Accumulate.



Decode fields op	Instruction Details
0	CADD
1	SQCADD

SVE2 integer absolute difference and accumulate long

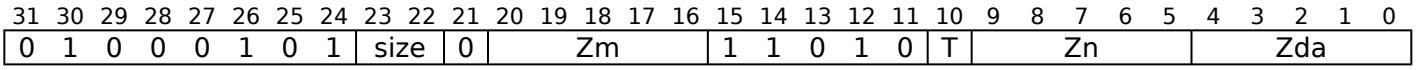
These instructions are under SVE2 Accumulate.



Decode fields U	T	Instruction Details
0	0	SABALB
0	1	SABALT
1	0	UABALB
1	1	UABALT

SVE2 integer add/subtract long with carry

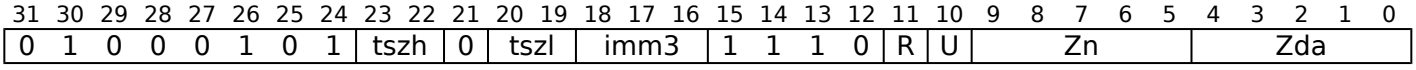
These instructions are under SVE2 Accumulate.



Decode fields size	T	Instruction Details
0x	0	ADCLB
0x	1	ADCLT
1x	0	SBCLB
1x	1	SBCLT

SVE2 bitwise shift right and accumulate

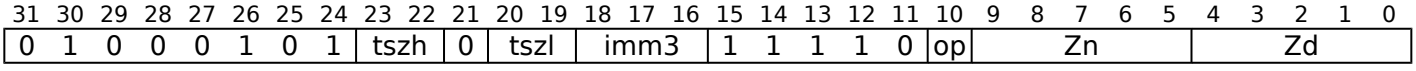
These instructions are under SVE2 Accumulate.



Decode fields R	U	Instruction Details
0	0	SSRA
0	1	USRA
1	0	SRSRA
1	1	URSRA

SVE2 bitwise shift and insert

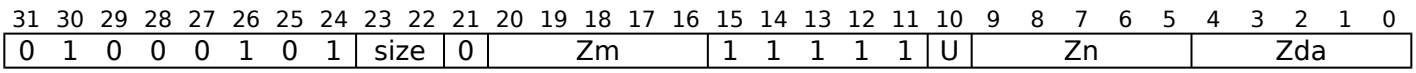
These instructions are under SVE2 Accumulate.



Decode fields op	Instruction Details
0	SRI
1	SLI

SVE2 integer absolute difference and accumulate

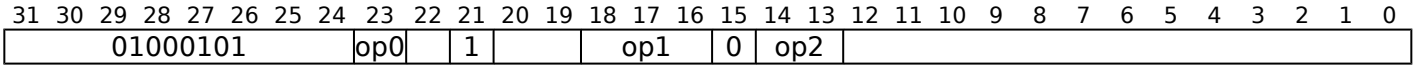
These instructions are under SVE2 Accumulate.



Decode fields U	Instruction Details
0	SABA
1	UABA

SVE2 Narrowing

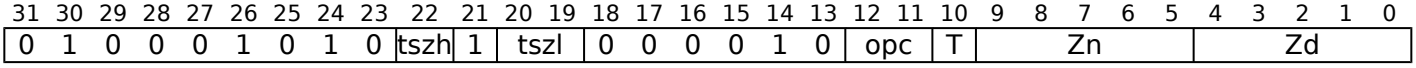
These instructions are under SVE encodings.



Decode fields op0	op1	op2	Instruction details
0	000	10	SVE2 saturating extract narrow
0	!= 000	10	UNALLOCATED
0		0x	SVE2 bitwise shift right narrow
1		0x	UNALLOCATED
1		10	UNALLOCATED
		11	SVE2 integer add/subtract narrow high part

SVE2 saturating extract narrow

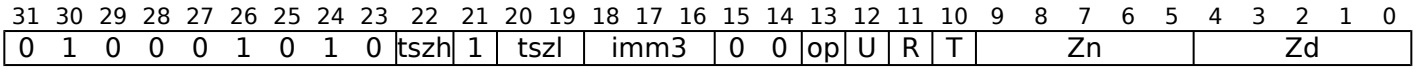
These instructions are under SVE2 Narrowing.



Decode fields opc	T	Instruction Details
00	0	SQXTNB
00	1	SQXTNT
01	0	UQXTNB
01	1	UQXTNT
10	0	SQXTUNB
10	1	SQXTUNT
11		UNALLOCATED

SVE2 bitwise shift right narrow

These instructions are under SVE2 Narrowing.

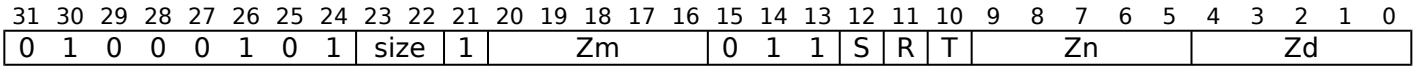


Decode fields op	U	R	T	Instruction Details
0	0	0	0	SQSHRUNB
0	0	0	1	SQSHRUNT
0	0	1	0	SQRSHRUNB
0	0	1	1	SQRSHRUNT
0	1	0	0	SHRNB
0	1	0	1	SHRNT
0	1	1	0	RSHRNB

Decode fields op	U	R	T	Instruction Details
0	1	1	1	RSHRNT
1	0	0	0	SQSHRNB
1	0	0	1	SQSHRNT
1	0	1	0	SQRSHRNB
1	0	1	1	SQRSHRNT
1	1	0	0	UQSHRNB
1	1	0	1	UQSHRNT
1	1	1	0	UQRSHRNB
1	1	1	1	UQRSHRNT

SVE2 integer add/subtract narrow high part

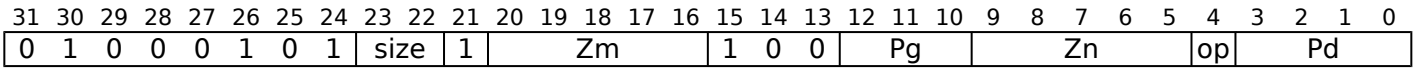
These instructions are under SVE2 Narrowing.



Decode fields S	R	T	Instruction Details
0	0	0	ADDHNB
0	0	1	ADDHNT
0	1	0	RADDHNB
0	1	1	RADDHNT
1	0	0	SUBHNB
1	0	1	SUBHNT
1	1	0	RSUBHNB
1	1	1	RSUBHNT

SVE2 character match

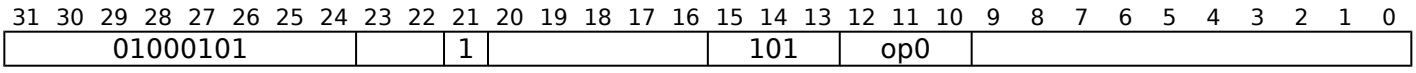
These instructions are under SVE encodings.



Decode fields op	Instruction Details
0	MATCH
1	NMATCH

SVE2 Histogram Computation - Segment

These instructions are under SVE encodings.



Decode fields op0	Instruction details
000	HISTSEG
!= 000	UNALLOCATED

SVE2 Crypto Extensions

These instructions are under SVE encodings.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01000101										1	op0			op1		111			op2			op3									

Decode fields op0	op1	op2	op3	Instruction details
000	00	00	00000	SVE2 crypto unary operations
000	00	00	!= 00000	UNALLOCATED
000	00	x1		UNALLOCATED
000	01	0x		UNALLOCATED
000	01	11		UNALLOCATED
000	1x	00		SVE2 crypto destructive binary operations
000	1x	x1		UNALLOCATED
!= 000		0x		UNALLOCATED
!= 000		11		UNALLOCATED
		10		SVE2 crypto constructive binary operations

SVE2 crypto unary operations

These instructions are under SVE2 Crypto Extensions.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	size		1	0	0	0	0	0	1	1	1	0	0	op	0	0	0	0	0	Zdn				

Decode fields size	op	Instruction Details	Feature
00	0	AESMC	FEAT_SVE_AES
00	1	AESIMC	FEAT_SVE_AES
01		UNALLOCATED	-
1x		UNALLOCATED	-

SVE2 crypto destructive binary operations

These instructions are under SVE2 Crypto Extensions.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	size		1	0	0	0	1	op	1	1	1	0	0	o2	Zm					Zdn				

Decode fields size	op	o2	Instruction Details	Feature
00	0	0	AESE	FEAT_SVE_AES
00	0	1	AESD	FEAT_SVE_AES
00	1	0	SM4E	FEAT_SVE_SM4
00	1	1	UNALLOCATED	-
01			UNALLOCATED	-
1x			UNALLOCATED	-

SVE2 crypto constructive binary operations

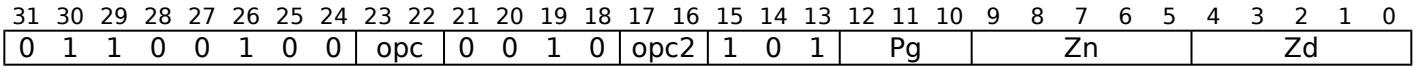
These instructions are under SVE2 Crypto Extensions.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	size		1	Zm					1	1	1	1	0	op	Zn					Zd				

Decode fields size	op	Instruction Details	Feature
00	0	SM4EKEY	FEAT_SVE_SM4
00	1	RAX1	FEAT_SVE_SHA3
01		UNALLOCATED	-
1x		UNALLOCATED	-

SVE floating-point convert precision odd elements

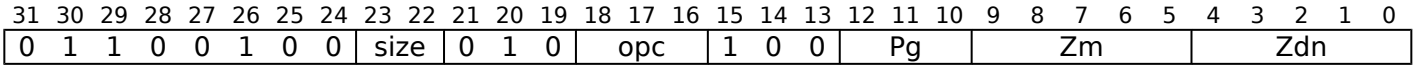
These instructions are under SVE encodings.



Decode fields opc	opc2	Instruction Details	Feature
x0	11	UNALLOCATED	-
00	0x	UNALLOCATED	-
00	10	FCVTXNT	-
01		UNALLOCATED	-
10	00	FCVTNT — single-precision to half-precision	-
10	01	FCVTLT — half-precision to single-precision	-
10	10	BFCVTNT	FEAT_BF16
11	0x	UNALLOCATED	-
11	10	FCVTNT — double-precision to single-precision	-
11	11	FCVTLT — single-precision to double-precision	-

SVE2 floating-point pairwise operations

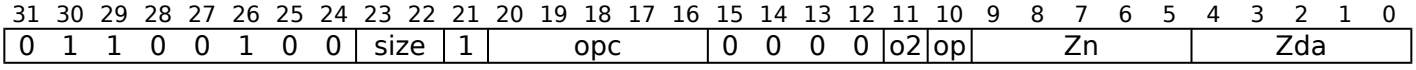
These instructions are under SVE encodings.



Decode fields opc	Instruction Details
000	FADDP
001	UNALLOCATED
01x	UNALLOCATED
100	FMAXNMP
101	FMINNMP
110	FMAXP
111	FMINP

SVE floating-point multiply-add (indexed)

These instructions are under SVE encodings.

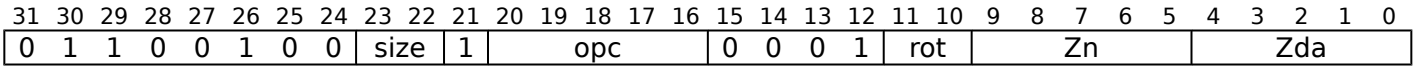


Decode fields size	o2	op	Instruction Details
	1		UNALLOCATED

Decode fields size	o2	op	Instruction Details
0x	0	0	FMLA (indexed) — half-precision
0x	0	1	FMLS (indexed) — half-precision
10	0	0	FMLA (indexed) — single-precision
10	0	1	FMLS (indexed) — single-precision
11	0	0	FMLA (indexed) — double-precision
11	0	1	FMLS (indexed) — double-precision

SVE floating-point complex multiply-add (indexed)

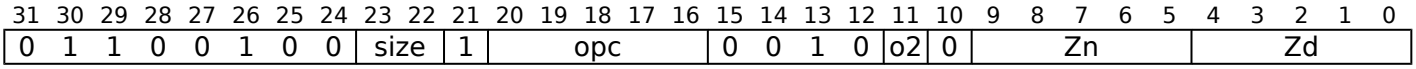
These instructions are under SVE encodings.



Decode fields size	Instruction Details
0x	UNALLOCATED
10	FCMLA (indexed) — half-precision
11	FCMLA (indexed) — single-precision

SVE floating-point multiply (indexed)

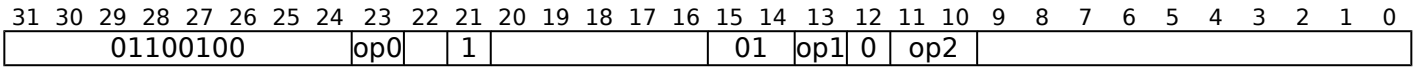
These instructions are under SVE encodings.



Decode fields size	o2	Instruction Details
	1	UNALLOCATED
0x	0	FMUL (indexed) — half-precision
10	0	FMUL (indexed) — single-precision
11	0	FMUL (indexed) — double-precision

SVE Floating Point Widening Multiply-Add - Indexed

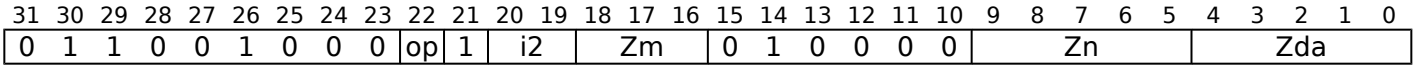
These instructions are under SVE encodings.



Decode fields op0	op1	op2	Instruction details
0	0	00	SVE BFloat16 floating-point dot product (indexed)
0	0	!= 00	UNALLOCATED
0	1		UNALLOCATED
1			SVE floating-point multiply-add long (indexed)

SVE BFloat16 floating-point dot product (indexed)

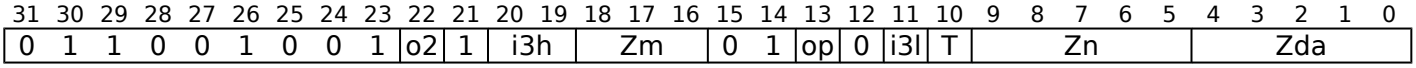
These instructions are under SVE Floating Point Widening Multiply-Add - Indexed.



Decode fields op	Instruction Details	Feature
0	UNALLOCATED	-
1	BFDOT (indexed)	FEAT_BF16

SVE floating-point multiply-add long (indexed)

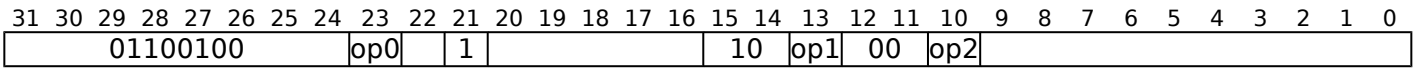
These instructions are under SVE Floating Point Widening Multiply-Add - Indexed.



Decode fields o2	op	T	Instruction Details	Feature
0	0	0	FMLALB (indexed)	-
0	0	1	FMLALT (indexed)	-
0	1	0	FMLSLB (indexed)	-
0	1	1	FMLSLT (indexed)	-
1	0	0	BFMLALB (indexed)	FEAT_BF16
1	0	1	BFMLALT (indexed)	FEAT_BF16
1	1		UNALLOCATED	-

SVE Floating Point Widening Multiply-Add

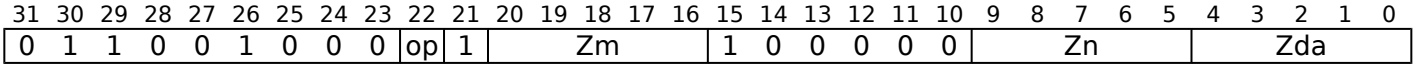
These instructions are under SVE encodings.



Decode fields op0	op1	op2	Instruction details
0	0	0	SVE BFloat16 floating-point dot product
0	0	1	UNALLOCATED
0	1		UNALLOCATED
1			SVE floating-point multiply-add long

SVE BFloat16 floating-point dot product

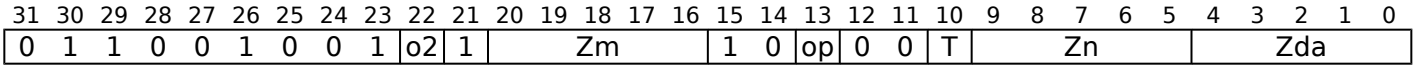
These instructions are under SVE Floating Point Widening Multiply-Add.



Decode fields op	Instruction Details	Feature
0	UNALLOCATED	-
1	BFDOT (vectors)	FEAT_BF16

SVE floating-point multiply-add long

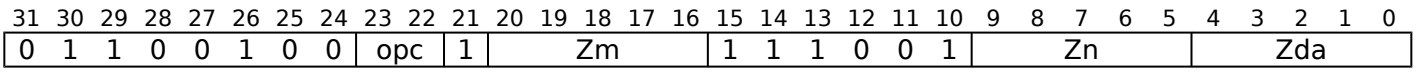
These instructions are under SVE Floating Point Widening Multiply-Add.



Decode fields o2	op	T	Instruction Details	Feature
0	0	0	FMLALB (vectors)	-
0	0	1	FMLALT (vectors)	-
0	1	0	FMLSLB (vectors)	-
0	1	1	FMLSLT (vectors)	-
1	0	0	BFMLALB (vectors)	FEAT_BF16
1	0	1	BFMLALT (vectors)	FEAT_BF16
1	1		UNALLOCATED	-

SVE floating point matrix multiply accumulate

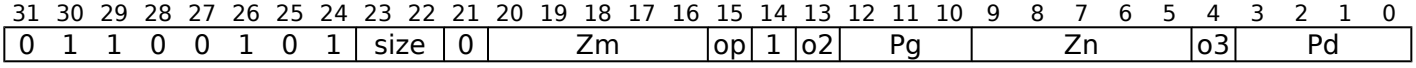
These instructions are under SVE encodings.



Decode fields opc	Instruction Details	Feature
00	UNALLOCATED	-
01	BFMMLA	FEAT_BF16
10	FMMLA — 32-bit element	FEAT_F32MM
11	FMMLA — 64-bit element	FEAT_F64MM

SVE floating-point compare vectors

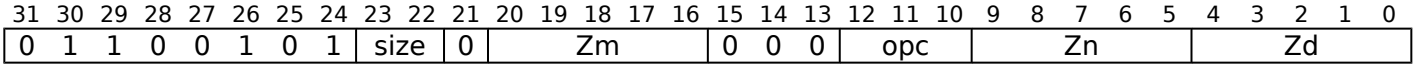
These instructions are under SVE encodings.



Decode fields op	o2	o3	Instruction Details
0	0	0	FCM<cc> (vectors) — FCMGE
0	0	1	FCM<cc> (vectors) — FCMGT
0	1	0	FCM<cc> (vectors) — FCMEQ
0	1	1	FCM<cc> (vectors) — FCMNE
1	0	0	FCM<cc> (vectors) — FCMUO
1	0	1	FAC<cc> — FACGE
1	1	0	UNALLOCATED
1	1	1	FAC<cc> — FACGT

SVE floating-point arithmetic (unpredicated)

These instructions are under SVE encodings.

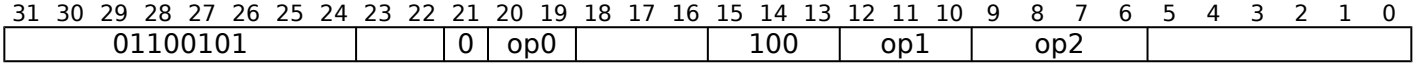


Decode fields opc	Instruction Details
000	FADD (vectors, unpredicated)

Decode fields opc	Instruction Details
001	FSUB (vectors, unpredicated)
010	FMUL (vectors, unpredicated)
011	FTSMUL
10x	UNALLOCATED
110	FRECPS
111	FRSQRTS

SVE Floating Point Arithmetic - Predicated

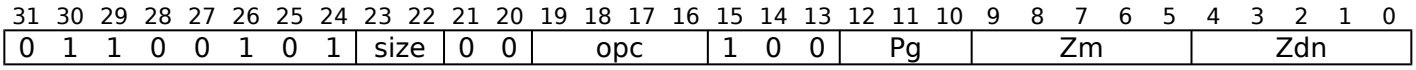
These instructions are under SVE encodings.



op0	Decode fields op1	op2	Instruction details
0x			SVE floating-point arithmetic (predicated)
10	000		FTMAD
10	!= 000		UNALLOCATED
11		0000	SVE floating-point arithmetic with immediate (predicated)
11		!= 0000	UNALLOCATED

SVE floating-point arithmetic (predicated)

These instructions are under SVE Floating Point Arithmetic - Predicated.



Decode fields opc	Instruction Details
0000	FADD (vectors, predicated)
0001	FSUB (vectors, predicated)
0010	FMUL (vectors, predicated)
0011	FSUBR (vectors)
0100	FMAXNM (vectors)
0101	FMINNM (vectors)
0110	FMAX (vectors)
0111	FMIN (vectors)
1000	FABD
1001	FSCALE
1010	FMULX
1011	UNALLOCATED
1100	FDIVR
1101	FDIV
111x	UNALLOCATED

SVE floating-point arithmetic with immediate (predicated)

These instructions are under SVE Floating Point Arithmetic - Predicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size		0	1	1	opc			1	0	0	Pg			0	0	0	0	i1	Zdn				

Decode fields opc	Instruction Details
000	FADD (immediate)
001	FSUB (immediate)
010	FMUL (immediate)
011	FSUBR (immediate)
100	FMAXNM (immediate)
101	FMINNM (immediate)
110	FMAX (immediate)
111	FMIN (immediate)

SVE Floating Point Unary Operations - Predicated

These instructions are under SVE encodings.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
01100101										0	op0					101															

Decode fields op0	Instruction details
00x	SVE floating-point round to integral value
010	SVE floating-point convert precision
011	SVE floating-point unary operations
10x	SVE integer convert to floating-point
11x	SVE floating-point convert to integer

SVE floating-point round to integral value

These instructions are under SVE Floating Point Unary Operations - Predicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size		0	0	0	opc			1	0	1	Pg			Zn					Zd				

Decode fields opc	Instruction Details
000	FRINT<r> — nearest with ties to even
001	FRINT<r> — toward plus infinity
010	FRINT<r> — toward minus infinity
011	FRINT<r> — toward zero
100	FRINT<r> — nearest with ties to away
101	UNALLOCATED
110	FRINT<r> — current mode signalling inexact
111	FRINT<r> — current mode

SVE floating-point convert precision

These instructions are under SVE Floating Point Unary Operations - Predicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	opc		0	0	1	0	opc2		1	0	1	Pg			Zn					Zd				

Decode fields opc	opc2	Instruction Details	Feature
x0	11	UNALLOCATED	-
00	0x	UNALLOCATED	-
00	10	FCVTX	-
01		UNALLOCATED	-
10	00	FCVT — single-precision to half-precision	-
10	01	FCVT — half-precision to single-precision	-
10	10	BFCVT	FEAT_BF16
11	00	FCVT — double-precision to half-precision	-
11	01	FCVT — half-precision to double-precision	-
11	10	FCVT — double-precision to single-precision	-
11	11	FCVT — single-precision to double-precision	-

SVE floating-point unary operations

These instructions are under SVE Floating Point Unary Operations - Predicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size		0	0	1	1	opc		1	0	1	Pg			Zn					Zd				

Decode fields opc	Instruction Details
00	FRECPX
01	FSQRT
1x	UNALLOCATED

SVE integer convert to floating-point

These instructions are under SVE Floating Point Unary Operations - Predicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	opc		0	1	0	opc2		U	1	0	1	Pg			Zn					Zd				

Decode fields opc	opc2	U	Instruction Details
00			UNALLOCATED
01	00		UNALLOCATED
01	01	0	SCVTF — 16-bit to half-precision
01	01	1	UCVTF — 16-bit to half-precision
01	10	0	SCVTF — 32-bit to half-precision
01	10	1	UCVTF — 32-bit to half-precision
01	11	0	SCVTF — 64-bit to half-precision
01	11	1	UCVTF — 64-bit to half-precision
10	0x		UNALLOCATED
10	10	0	SCVTF — 32-bit to single-precision
10	10	1	UCVTF — 32-bit to single-precision
10	11		UNALLOCATED
11	00	0	SCVTF — 32-bit to double-precision
11	00	1	UCVTF — 32-bit to double-precision
11	01		UNALLOCATED
11	10	0	SCVTF — 64-bit to single-precision

Decode fields opc	opc2	U	Instruction Details
11	10	1	UCVTF — 64-bit to single-precision
11	11	0	SCVTF — 64-bit to double-precision
11	11	1	UCVTF — 64-bit to double-precision

SVE floating-point convert to integer

These instructions are under SVE Floating Point Unary Operations - Predicated.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	opc		0	1	1	opc2		U	1	0	1	Pg			Zn					Zd				

Decode fields opc	opc2	U	Instruction Details
00		0	FLOGB
00		1	UNALLOCATED
01	00		UNALLOCATED
01	01	0	FCVTZS — half-precision to 16-bit
01	01	1	FCVTZU — half-precision to 16-bit
01	10	0	FCVTZS — half-precision to 32-bit
01	10	1	FCVTZU — half-precision to 32-bit
01	11	0	FCVTZS — half-precision to 64-bit
01	11	1	FCVTZU — half-precision to 64-bit
10	0x		UNALLOCATED
10	10	0	FCVTZS — single-precision to 32-bit
10	10	1	FCVTZU — single-precision to 32-bit
10	11		UNALLOCATED
11	00	0	FCVTZS — double-precision to 32-bit
11	00	1	FCVTZU — double-precision to 32-bit
11	01		UNALLOCATED
11	10	0	FCVTZS — single-precision to 64-bit
11	10	1	FCVTZU — single-precision to 64-bit
11	11	0	FCVTZS — double-precision to 64-bit
11	11	1	FCVTZU — double-precision to 64-bit

SVE floating-point recursive reduction

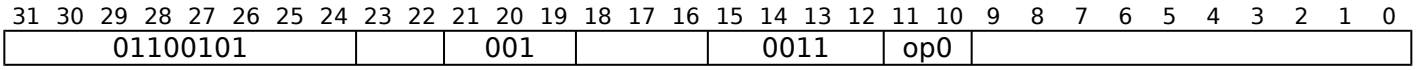
These instructions are under SVE encodings.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	1	size		0	0	0	opc			0	0	1	Pg			Zn					Vd				

Decode fields opc	Instruction Details
000	FADDV
001	UNALLOCATED
01x	UNALLOCATED
100	FMAXNMV
101	FMINNMV
110	FMAXV
111	FMINV

SVE Floating Point Unary Operations - Unpredicated

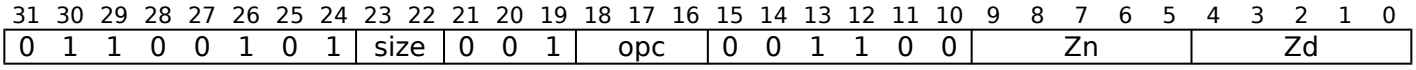
These instructions are under SVE encodings.



Decode fields op0	Instruction details
00	SVE floating-point reciprocal estimate (unpredicated)
!= 00	UNALLOCATED

SVE floating-point reciprocal estimate (unpredicated)

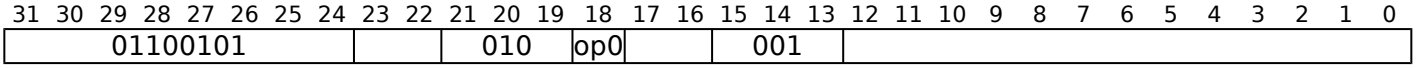
These instructions are under SVE Floating Point Unary Operations - Unpredicated.



Decode fields opc	Instruction Details
0xx	UNALLOCATED
10x	UNALLOCATED
110	FRECPE
111	FRSQRTE

SVE Floating Point Compare - with Zero

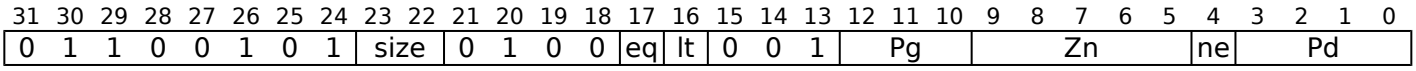
These instructions are under SVE encodings.



Decode fields op0	Instruction details
0	SVE floating-point compare with zero
1	UNALLOCATED

SVE floating-point compare with zero

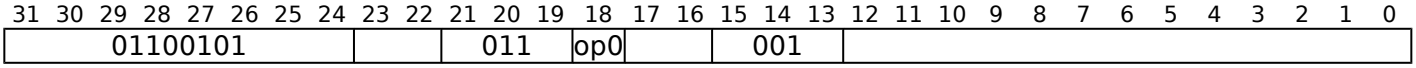
These instructions are under SVE Floating Point Compare - with Zero.



Decode fields eq	lt	ne	Instruction Details
0	0	0	FCM<cc> (zero) — FCMGE
0	0	1	FCM<cc> (zero) — FCMGT
0	1	0	FCM<cc> (zero) — FCMLT
0	1	1	FCM<cc> (zero) — FCMLE
1		1	UNALLOCATED
1	0	0	FCM<cc> (zero) — FCMEQ
1	1	0	FCM<cc> (zero) — FCMNE

SVE Floating Point Accumulating Reduction

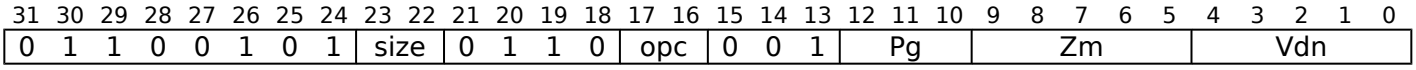
These instructions are under SVE encodings.



Decode fields op0	Instruction details
0	SVE floating-point serial reduction (predicated)
1	UNALLOCATED

SVE floating-point serial reduction (predicated)

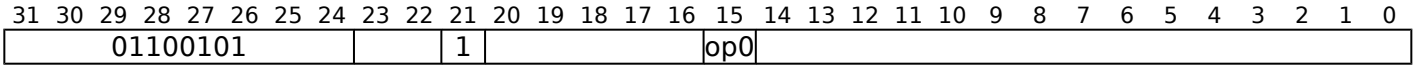
These instructions are under SVE Floating Point Accumulating Reduction.



Decode fields opc	Instruction Details
00	FADDA
01	UNALLOCATED
1x	UNALLOCATED

SVE Floating Point Multiply-Add

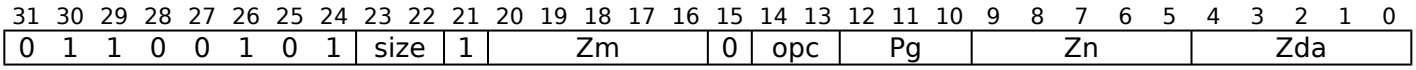
These instructions are under SVE encodings.



Decode fields op0	Instruction details
0	SVE floating-point multiply-accumulate writing addend
1	SVE floating-point multiply-accumulate writing multiplicand

SVE floating-point multiply-accumulate writing addend

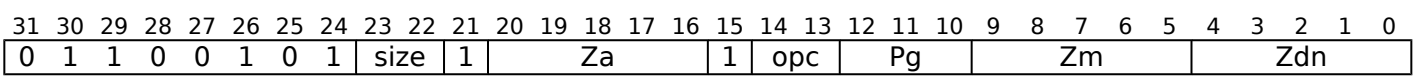
These instructions are under SVE Floating Point Multiply-Add.



Decode fields opc	Instruction Details
00	FMLA (vectors)
01	FMLS (vectors)
10	FNMLA
11	FNMLS

SVE floating-point multiply-accumulate writing multiplicand

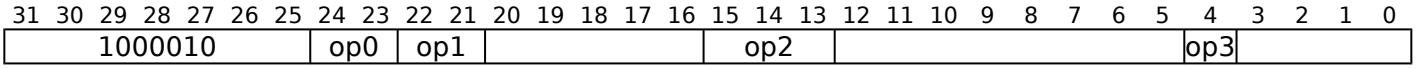
These instructions are under SVE Floating Point Multiply-Add.



Decode fields opc	Instruction Details
00	FMAD
01	FMSB
10	FNMAD
11	FNMSB

SVE Memory - 32-bit Gather and Unsized Contiguous

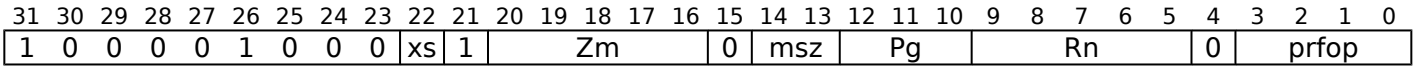
These instructions are under SVE encodings.



Decode fields op0	op1	op2	op3	Instruction details
00	x1	0xx	0	SVE 32-bit gather prefetch (scalar plus 32-bit scaled offsets)
00	x1	0xx	1	UNALLOCATED
01	x1	0xx		SVE 32-bit gather load halfwords (scalar plus 32-bit scaled offsets)
10	x1	0xx		SVE 32-bit gather load words (scalar plus 32-bit scaled offsets)
11	0x	000	0	LDR (predicate)
11	0x	000	1	UNALLOCATED
11	0x	010		LDR (vector)
11	0x	0x1		UNALLOCATED
11	1x	0xx	0	SVE contiguous prefetch (scalar plus immediate)
11	1x	0xx	1	UNALLOCATED
!= 11	x0	0xx		SVE 32-bit gather load (scalar plus 32-bit unscaled offsets)
	00	10x		SVE2 32-bit gather non-temporal load (vector plus scalar)
	00	110	0	SVE contiguous prefetch (scalar plus scalar)
	00	111	0	SVE 32-bit gather prefetch (vector plus immediate)
	00	11x	1	UNALLOCATED
	01	1xx		SVE 32-bit gather load (vector plus immediate)
	1x	1xx		SVE load and broadcast element

SVE 32-bit gather prefetch (scalar plus 32-bit scaled offsets)

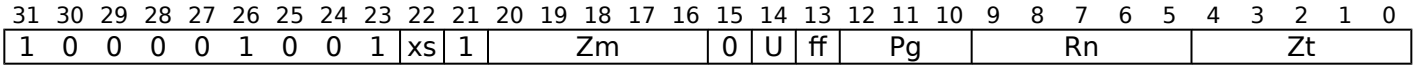
These instructions are under SVE Memory - 32-bit Gather and Unsized Contiguous.



Decode fields msz	Instruction Details
00	PRFB (scalar plus vector)
01	PRFH (scalar plus vector)
10	PRFW (scalar plus vector)
11	PRFD (scalar plus vector)

SVE 32-bit gather load halfwords (scalar plus 32-bit scaled offsets)

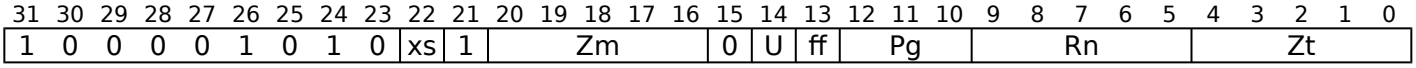
These instructions are under SVE Memory - 32-bit Gather and Unsized Contiguous.



Decode fields U	ff	Instruction Details
0	0	LD1SH (scalar plus vector)
0	1	LDFF1SH (scalar plus vector)
1	0	LD1H (scalar plus vector)
1	1	LDFF1H (scalar plus vector)

SVE 32-bit gather load words (scalar plus 32-bit scaled offsets)

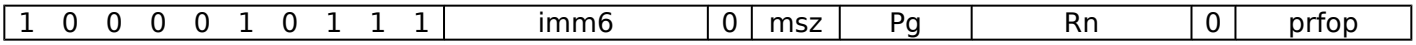
These instructions are under SVE Memory - 32-bit Gather and Unsized Contiguous.



Decode fields U	ff	Instruction Details
0		UNALLOCATED
1	0	LD1W (scalar plus vector)
1	1	LDFF1W (scalar plus vector)

SVE contiguous prefetch (scalar plus immediate)

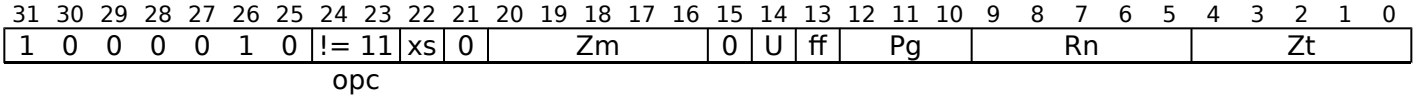
These instructions are under SVE Memory - 32-bit Gather and Unsized Contiguous.



Decode fields msz	Instruction Details
00	PRFB (scalar plus immediate)
01	PRFH (scalar plus immediate)
10	PRFW (scalar plus immediate)
11	PRFD (scalar plus immediate)

SVE 32-bit gather load (scalar plus 32-bit unscaled offsets)

These instructions are under SVE Memory - 32-bit Gather and Unsized Contiguous.



The following constraints also apply to this encoding: opc != 11 && opc != 11

Decode fields opc	U	ff	Instruction Details
00	0	0	LD1SB (scalar plus vector)
00	0	1	LDFF1SB (scalar plus vector)
00	1	0	LD1B (scalar plus vector)
00	1	1	LDFF1B (scalar plus vector)
01	0	0	LD1SH (scalar plus vector)
01	0	1	LDFF1SH (scalar plus vector)
01	1	0	LD1H (scalar plus vector)

Decode fields opc	U	ff	Instruction Details
01	1	1	LDFF1H (scalar plus vector)
10	0		UNALLOCATED
10	1	0	LD1W (scalar plus vector)
10	1	1	LDFF1W (scalar plus vector)

SVE2 32-bit gather non-temporal load (vector plus scalar)

These instructions are under SVE Memory - 32-bit Gather and Unsized Contiguous.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	msz		0	0	Rm					1	0	U	Pg			Zn					Zt				

Decode fields msz	U	Instruction Details
00	0	LDNT1SB
00	1	LDNT1B (vector plus scalar)
01	0	LDNT1SH
01	1	LDNT1H (vector plus scalar)
10	0	UNALLOCATED
10	1	LDNT1W (vector plus scalar)
11		UNALLOCATED

SVE contiguous prefetch (scalar plus scalar)

These instructions are under SVE Memory - 32-bit Gather and Unsized Contiguous.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	msz		0	0	Rm					1	1	0	Pg			Rn					0	prfop			

Decode fields msz	Instruction Details
00	PRFB (scalar plus scalar)
01	PRFH (scalar plus scalar)
10	PRFW (scalar plus scalar)
11	PRFD (scalar plus scalar)

SVE 32-bit gather prefetch (vector plus immediate)

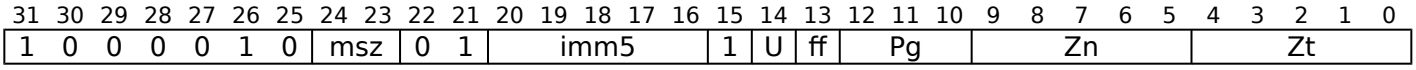
These instructions are under SVE Memory - 32-bit Gather and Unsized Contiguous.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	msz		0	0	imm5					1	1	1	Pg			Zn					0	prfop			

Decode fields msz	Instruction Details
00	PRFB (vector plus immediate)
01	PRFH (vector plus immediate)
10	PRFW (vector plus immediate)
11	PRFD (vector plus immediate)

SVE 32-bit gather load (vector plus immediate)

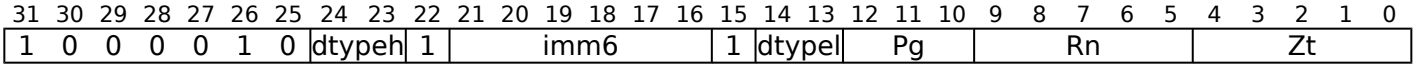
These instructions are under SVE Memory - 32-bit Gather and Unsized Contiguous.



Decode fields msz	U	ff	Instruction Details
00	0	0	LD1SB (vector plus immediate)
00	0	1	LDFF1SB (vector plus immediate)
00	1	0	LD1B (vector plus immediate)
00	1	1	LDFF1B (vector plus immediate)
01	0	0	LD1SH (vector plus immediate)
01	0	1	LDFF1SH (vector plus immediate)
01	1	0	LD1H (vector plus immediate)
01	1	1	LDFF1H (vector plus immediate)
10	0		UNALLOCATED
10	1	0	LD1W (vector plus immediate)
10	1	1	LDFF1W (vector plus immediate)
11			UNALLOCATED

SVE load and broadcast element

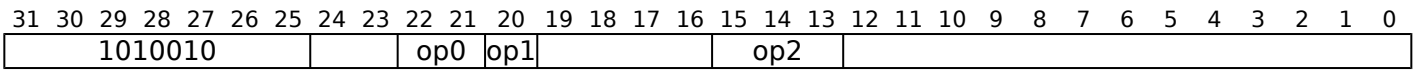
These instructions are under SVE Memory - 32-bit Gather and Unsized Contiguous.



Decode fields dtypeh	dtypel	Instruction Details
00	00	LD1RB — 8-bit element
00	01	LD1RB — 16-bit element
00	10	LD1RB — 32-bit element
00	11	LD1RB — 64-bit element
01	00	LD1RSW
01	01	LD1RH — 16-bit element
01	10	LD1RH — 32-bit element
01	11	LD1RH — 64-bit element
10	00	LD1RSH — 64-bit element
10	01	LD1RSH — 32-bit element
10	10	LD1RW — 32-bit element
10	11	LD1RW — 64-bit element
11	00	LD1RSB — 64-bit element
11	01	LD1RSB — 32-bit element
11	10	LD1RSB — 16-bit element
11	11	LD1RD

SVE Memory - Contiguous Load

These instructions are under SVE encodings.



Decode fields op0	op1	op2	Instruction details
00	0	111	SVE contiguous non-temporal load (scalar plus immediate)

00		110	SVE contiguous non-temporal load (scalar plus scalar)
!= 00	0	111	SVE load multiple structures (scalar plus immediate)
!= 00		110	SVE load multiple structures (scalar plus scalar)
	0	001	SVE load and broadcast quadword (scalar plus immediate)
	0	101	SVE contiguous load (scalar plus immediate)
	1	001	UNALLOCATED
	1	101	SVE contiguous non-fault load (scalar plus immediate)
	1	111	UNALLOCATED
		000	SVE load and broadcast quadword (scalar plus scalar)
		010	SVE contiguous load (scalar plus scalar)
		011	SVE contiguous first-fault load (scalar plus scalar)
		100	UNALLOCATED

SVE contiguous non-temporal load (scalar plus immediate)

These instructions are under SVE Memory - Contiguous Load.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	msz		0	0	0	imm4				1	1	1	Pg			Rn					Zt				

Decode fields msz	Instruction Details
00	LDNT1B (scalar plus immediate)
01	LDNT1H (scalar plus immediate)
10	LDNT1W (scalar plus immediate)
11	LDNT1D (scalar plus immediate)

SVE contiguous non-temporal load (scalar plus scalar)

These instructions are under SVE Memory - Contiguous Load.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	msz		0	0	Rm					1	1	0	Pg			Rn					Zt				

Decode fields msz	Instruction Details
00	LDNT1B (scalar plus scalar)
01	LDNT1H (scalar plus scalar)
10	LDNT1W (scalar plus scalar)
11	LDNT1D (scalar plus scalar)

SVE load multiple structures (scalar plus immediate)

These instructions are under SVE Memory - Contiguous Load.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	msz		!= 00		0	imm4				1	1	1	Pg			Rn					Zt				
									opc																						

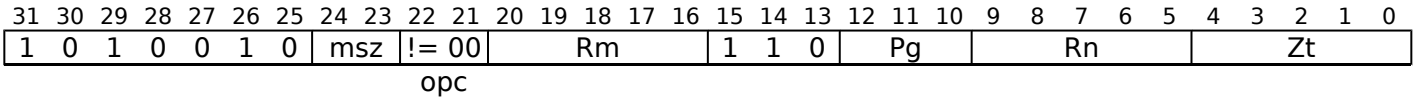
The following constraints also apply to this encoding: opc != 00 && opc != 00

Decode fields msz	opc	Instruction Details
00	01	LD2B (scalar plus immediate)

Decode fields msz	opc	Instruction Details
00	10	LD3B (scalar plus immediate)
00	11	LD4B (scalar plus immediate)
01	01	LD2H (scalar plus immediate)
01	10	LD3H (scalar plus immediate)
01	11	LD4H (scalar plus immediate)
10	01	LD2W (scalar plus immediate)
10	10	LD3W (scalar plus immediate)
10	11	LD4W (scalar plus immediate)
11	01	LD2D (scalar plus immediate)
11	10	LD3D (scalar plus immediate)
11	11	LD4D (scalar plus immediate)

SVE load multiple structures (scalar plus scalar)

These instructions are under SVE Memory - Contiguous Load.

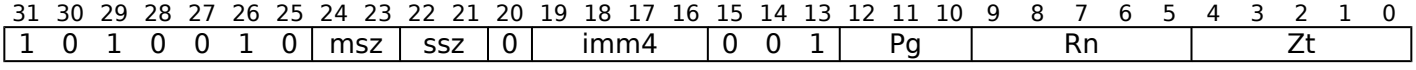


The following constraints also apply to this encoding: opc != 00 && opc != 00

Decode fields msz	opc	Instruction Details
00	01	LD2B (scalar plus scalar)
00	10	LD3B (scalar plus scalar)
00	11	LD4B (scalar plus scalar)
01	01	LD2H (scalar plus scalar)
01	10	LD3H (scalar plus scalar)
01	11	LD4H (scalar plus scalar)
10	01	LD2W (scalar plus scalar)
10	10	LD3W (scalar plus scalar)
10	11	LD4W (scalar plus scalar)
11	01	LD2D (scalar plus scalar)
11	10	LD3D (scalar plus scalar)
11	11	LD4D (scalar plus scalar)

SVE load and broadcast quadword (scalar plus immediate)

These instructions are under SVE Memory - Contiguous Load.



Decode fields msz	ssz	Instruction Details	Feature
	1x	UNALLOCATED	-
00	00	LD1RQB (scalar plus immediate)	-
00	01	LD1ROB (scalar plus immediate)	FEAT_F64MM
01	00	LD1RQH (scalar plus immediate)	-
01	01	LD1ROH (scalar plus immediate)	FEAT_F64MM

Decode fields msz	ssz	Instruction Details	Feature
10	00	LD1RQW (scalar plus immediate)	-
10	01	LD1ROW (scalar plus immediate)	FEAT_F64MM
11	00	LD1RQD (scalar plus immediate)	-
11	01	LD1ROD (scalar plus immediate)	FEAT_F64MM

SVE contiguous load (scalar plus immediate)

These instructions are under SVE Memory - Contiguous Load.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	dtype				0	imm4				1	0	1	Pg			Rn					Zt				

Decode fields dtype	Instruction Details
0000	LD1B (scalar plus immediate) — 8-bit element
0001	LD1B (scalar plus immediate) — 16-bit element
0010	LD1B (scalar plus immediate) — 32-bit element
0011	LD1B (scalar plus immediate) — 64-bit element
0100	LD1SW (scalar plus immediate)
0101	LD1H (scalar plus immediate) — 16-bit element
0110	LD1H (scalar plus immediate) — 32-bit element
0111	LD1H (scalar plus immediate) — 64-bit element
1000	LD1SH (scalar plus immediate) — 64-bit element
1001	LD1SH (scalar plus immediate) — 32-bit element
1010	LD1W (scalar plus immediate) — 32-bit element
1011	LD1W (scalar plus immediate) — 64-bit element
1100	LD1SB (scalar plus immediate) — 64-bit element
1101	LD1SB (scalar plus immediate) — 32-bit element
1110	LD1SB (scalar plus immediate) — 16-bit element
1111	LD1D (scalar plus immediate)

SVE contiguous non-fault load (scalar plus immediate)

These instructions are under SVE Memory - Contiguous Load.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	dtype				1	imm4				1	0	1	Pg			Rn					Zt				

Decode fields dtype	Instruction Details
0000	LDNF1B — 8-bit element
0001	LDNF1B — 16-bit element
0010	LDNF1B — 32-bit element
0011	LDNF1B — 64-bit element
0100	LDNF1SW
0101	LDNF1H — 16-bit element
0110	LDNF1H — 32-bit element
0111	LDNF1H — 64-bit element
1000	LDNF1SH — 64-bit element
1001	LDNF1SH — 32-bit element

Decode fields dtype	Instruction Details
1010	LDNF1W — 32-bit element
1011	LDNF1W — 64-bit element
1100	LDNF1SB — 64-bit element
1101	LDNF1SB — 32-bit element
1110	LDNF1SB — 16-bit element
1111	LDNF1D

SVE load and broadcast quadword (scalar plus scalar)

These instructions are under SVE Memory - Contiguous Load.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	msz		ssz		Rm					0	0	0	Pg			Rn					Zt				

Decode fields msz	ssz	Instruction Details	Feature
	1x	UNALLOCATED	-
00	00	LD1RQB (scalar plus scalar)	-
00	01	LD1ROB (scalar plus scalar)	FEAT_F64MM
01	00	LD1RQH (scalar plus scalar)	-
01	01	LD1ROH (scalar plus scalar)	FEAT_F64MM
10	00	LD1RQW (scalar plus scalar)	-
10	01	LD1ROW (scalar plus scalar)	FEAT_F64MM
11	00	LD1RQD (scalar plus scalar)	-
11	01	LD1ROD (scalar plus scalar)	FEAT_F64MM

SVE contiguous load (scalar plus scalar)

These instructions are under SVE Memory - Contiguous Load.

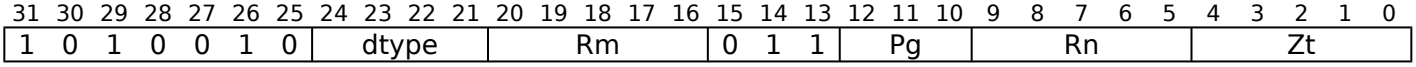
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	dtype				Rm					0	1	0	Pg			Rn					Zt				

Decode fields dtype	Instruction Details
0000	LD1B (scalar plus scalar) — 8-bit element
0001	LD1B (scalar plus scalar) — 16-bit element
0010	LD1B (scalar plus scalar) — 32-bit element
0011	LD1B (scalar plus scalar) — 64-bit element
0100	LD1SW (scalar plus scalar)
0101	LD1H (scalar plus scalar) — 16-bit element
0110	LD1H (scalar plus scalar) — 32-bit element
0111	LD1H (scalar plus scalar) — 64-bit element
1000	LD1SH (scalar plus scalar) — 64-bit element
1001	LD1SH (scalar plus scalar) — 32-bit element
1010	LD1W (scalar plus scalar) — 32-bit element
1011	LD1W (scalar plus scalar) — 64-bit element
1100	LD1SB (scalar plus scalar) — 64-bit element
1101	LD1SB (scalar plus scalar) — 32-bit element
1110	LD1SB (scalar plus scalar) — 16-bit element

Decode fields dtype	Instruction Details
1111	LD1D (scalar plus scalar)

SVE contiguous first-fault load (scalar plus scalar)

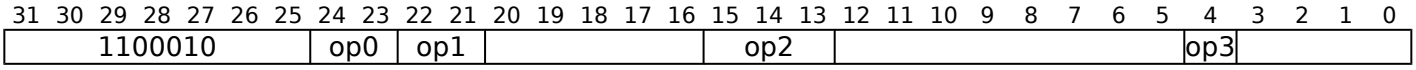
These instructions are under SVE Memory - Contiguous Load.



Decode fields dtype	Instruction Details
0000	LDFF1B (scalar plus scalar) — 8-bit element
0001	LDFF1B (scalar plus scalar) — 16-bit element
0010	LDFF1B (scalar plus scalar) — 32-bit element
0011	LDFF1B (scalar plus scalar) — 64-bit element
0100	LDFF1SW (scalar plus scalar)
0101	LDFF1H (scalar plus scalar) — 16-bit element
0110	LDFF1H (scalar plus scalar) — 32-bit element
0111	LDFF1H (scalar plus scalar) — 64-bit element
1000	LDFF1SH (scalar plus scalar) — 64-bit element
1001	LDFF1SH (scalar plus scalar) — 32-bit element
1010	LDFF1W (scalar plus scalar) — 32-bit element
1011	LDFF1W (scalar plus scalar) — 64-bit element
1100	LDFF1SB (scalar plus scalar) — 64-bit element
1101	LDFF1SB (scalar plus scalar) — 32-bit element
1110	LDFF1SB (scalar plus scalar) — 16-bit element
1111	LDFF1D (scalar plus scalar)

SVE Memory - 64-bit Gather

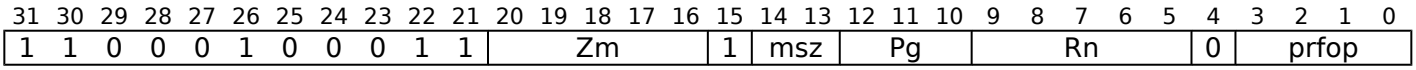
These instructions are under SVE encodings.



Decode fields op0	op1	op2	op3	Instruction details
00	01	0xx	1	UNALLOCATED
00	11	1xx	0	SVE 64-bit gather prefetch (scalar plus 64-bit scaled offsets)
00	11		1	UNALLOCATED
00	x1	0xx	0	SVE 64-bit gather prefetch (scalar plus unpacked 32-bit scaled offsets)
!= 00	11	1xx		SVE 64-bit gather load (scalar plus 64-bit scaled offsets)
!= 00	x1	0xx		SVE 64-bit gather load (scalar plus 32-bit unpacked scaled offsets)
	00	101		UNALLOCATED
	00	111	0	SVE 64-bit gather prefetch (vector plus immediate)
	00	111	1	UNALLOCATED
	00	1x0		SVE2 64-bit gather non-temporal load (vector plus scalar)
	01	1xx		SVE 64-bit gather load (vector plus immediate)
	10	1xx		SVE 64-bit gather load (scalar plus 64-bit unscaled offsets)
	x0	0xx		SVE 64-bit gather load (scalar plus unpacked 32-bit unscaled offsets)

SVE 64-bit gather prefetch (scalar plus 64-bit scaled offsets)

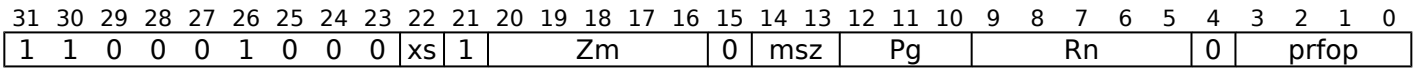
These instructions are under SVE Memory - 64-bit Gather.



Decode fields msz	Instruction Details
00	PRFB (scalar plus vector)
01	PRFH (scalar plus vector)
10	PRFW (scalar plus vector)
11	PRFD (scalar plus vector)

SVE 64-bit gather prefetch (scalar plus unpacked 32-bit scaled offsets)

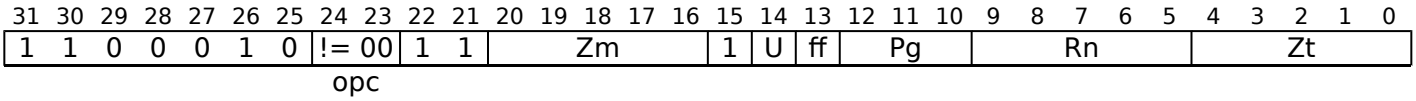
These instructions are under SVE Memory - 64-bit Gather.



Decode fields msz	Instruction Details
00	PRFB (scalar plus vector)
01	PRFH (scalar plus vector)
10	PRFW (scalar plus vector)
11	PRFD (scalar plus vector)

SVE 64-bit gather load (scalar plus 64-bit scaled offsets)

These instructions are under SVE Memory - 64-bit Gather.

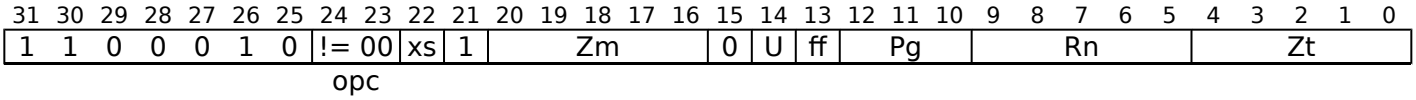


The following constraints also apply to this encoding: opc != 00 && opc != 00

Decode fields opc	U	ff	Instruction Details
01	0	0	LD1SH (scalar plus vector)
01	0	1	LDFF1SH (scalar plus vector)
01	1	0	LD1H (scalar plus vector)
01	1	1	LDFF1H (scalar plus vector)
10	0	0	LD1SW (scalar plus vector)
10	0	1	LDFF1SW (scalar plus vector)
10	1	0	LD1W (scalar plus vector)
10	1	1	LDFF1W (scalar plus vector)
11	0		UNALLOCATED
11	1	0	LD1D (scalar plus vector)
11	1	1	LDFF1D (scalar plus vector)

SVE 64-bit gather load (scalar plus 32-bit unpacked scaled offsets)

These instructions are under SVE Memory - 64-bit Gather.

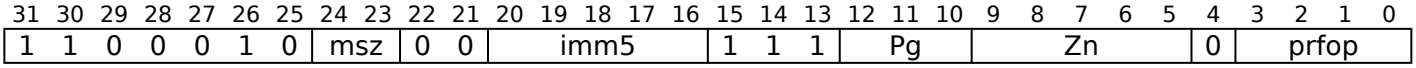


The following constraints also apply to this encoding: opc != 00 && opc != 00

Decode fields opc	U	ff	Instruction Details
01	0	0	LD1SH (scalar plus vector)
01	0	1	LDFF1SH (scalar plus vector)
01	1	0	LD1H (scalar plus vector)
01	1	1	LDFF1H (scalar plus vector)
10	0	0	LD1SW (scalar plus vector)
10	0	1	LDFF1SW (scalar plus vector)
10	1	0	LD1W (scalar plus vector)
10	1	1	LDFF1W (scalar plus vector)
11	0		UNALLOCATED
11	1	0	LD1D (scalar plus vector)
11	1	1	LDFF1D (scalar plus vector)

SVE 64-bit gather prefetch (vector plus immediate)

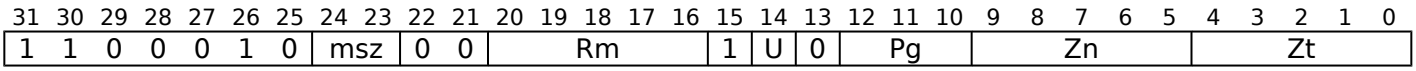
These instructions are under SVE Memory - 64-bit Gather.



Decode fields msz	Instruction Details
00	PRFB (vector plus immediate)
01	PRFH (vector plus immediate)
10	PRFW (vector plus immediate)
11	PRFD (vector plus immediate)

SVE2 64-bit gather non-temporal load (vector plus scalar)

These instructions are under SVE Memory - 64-bit Gather.



Decode fields msz	U	Instruction Details
00	0	LDNT1SB
00	1	LDNT1B (vector plus scalar)
01	0	LDNT1SH
01	1	LDNT1H (vector plus scalar)
10	0	LDNT1SW
10	1	LDNT1W (vector plus scalar)
11	0	UNALLOCATED
11	1	LDNT1D (vector plus scalar)

SVE 64-bit gather load (vector plus immediate)

These instructions are under SVE Memory - 64-bit Gather.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	msz		0	1	imm5					1	U	ff	Pg			Zn					Zt				

Decode fields msz	U	ff	Instruction Details
00	0	0	LD1SB (vector plus immediate)
00	0	1	LDFF1SB (vector plus immediate)
00	1	0	LD1B (vector plus immediate)
00	1	1	LDFF1B (vector plus immediate)
01	0	0	LD1SH (vector plus immediate)
01	0	1	LDFF1SH (vector plus immediate)
01	1	0	LD1H (vector plus immediate)
01	1	1	LDFF1H (vector plus immediate)
10	0	0	LD1SW (vector plus immediate)
10	0	1	LDFF1SW (vector plus immediate)
10	1	0	LD1W (vector plus immediate)
10	1	1	LDFF1W (vector plus immediate)
11	0		UNALLOCATED
11	1	0	LD1D (vector plus immediate)
11	1	1	LDFF1D (vector plus immediate)

SVE 64-bit gather load (scalar plus 64-bit unscaled offsets)

These instructions are under SVE Memory - 64-bit Gather.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	msz		1	0	Zm					1	U	ff	Pg			Rn					Zt				

Decode fields msz	U	ff	Instruction Details
00	0	0	LD1SB (scalar plus vector)
00	0	1	LDFF1SB (scalar plus vector)
00	1	0	LD1B (scalar plus vector)
00	1	1	LDFF1B (scalar plus vector)
01	0	0	LD1SH (scalar plus vector)
01	0	1	LDFF1SH (scalar plus vector)
01	1	0	LD1H (scalar plus vector)
01	1	1	LDFF1H (scalar plus vector)
10	0	0	LD1SW (scalar plus vector)
10	0	1	LDFF1SW (scalar plus vector)
10	1	0	LD1W (scalar plus vector)
10	1	1	LDFF1W (scalar plus vector)
11	0		UNALLOCATED
11	1	0	LD1D (scalar plus vector)
11	1	1	LDFF1D (scalar plus vector)

SVE 64-bit gather load (scalar plus unpacked 32-bit unscaled offsets)

These instructions are under SVE Memory - 64-bit Gather.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	1	0	msz		xs	0	Zm					0	U	ff	Pg			Rn					Zt				

Decode fields msz	U	ff	Instruction Details
00	0	0	LD1SB (scalar plus vector)
00	0	1	LDFF1SB (scalar plus vector)
00	1	0	LD1B (scalar plus vector)
00	1	1	LDFF1B (scalar plus vector)
01	0	0	LD1SH (scalar plus vector)
01	0	1	LDFF1SH (scalar plus vector)
01	1	0	LD1H (scalar plus vector)
01	1	1	LDFF1H (scalar plus vector)
10	0	0	LD1SW (scalar plus vector)
10	0	1	LDFF1SW (scalar plus vector)
10	1	0	LD1W (scalar plus vector)
10	1	1	LDFF1W (scalar plus vector)
11	0		UNALLOCATED
11	1	0	LD1D (scalar plus vector)
11	1	1	LDFF1D (scalar plus vector)

SVE Memory - Contiguous Store and Unsized Contiguous

These instructions are under SVE encodings.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1110010							op0									0	op1	0									op2				

Decode fields op0	op1	op2	Instruction details
0xx	0		UNALLOCATED
10x	0		UNALLOCATED
110	0	0	STR (predicate)
110	0	1	UNALLOCATED
110	1		STR (vector)
111	0		UNALLOCATED
!= 110	1		SVE contiguous store (scalar plus scalar)

SVE contiguous store (scalar plus scalar)

These instructions are under SVE Memory - Contiguous Store and Unsized Contiguous.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	!= 110			o2	Rm					0	1	0	Pg			Rn					Zt				
							opc																								

The following constraints also apply to this encoding: opc != 110 && opc != 110

Decode fields opc	o2	Instruction Details
00x		ST1B (scalar plus scalar)
01x		ST1H (scalar plus scalar)
101		ST1W (scalar plus scalar)
111	0	UNALLOCATED
111	1	ST1D (scalar plus scalar)

SVE Memory - Non-temporal and Multi-register Store

These instructions are under SVE encodings.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1110010									op0							0	op1	1														

Decode fields op0	op1	Instruction details
00	0	SVE2 64-bit scatter non-temporal store (vector plus scalar)
00	1	SVE contiguous non-temporal store (scalar plus scalar)
10	0	SVE2 32-bit scatter non-temporal store (vector plus scalar)
!= 00	1	SVE store multiple structures (scalar plus scalar)
x1	0	UNALLOCATED

SVE2 64-bit scatter non-temporal store (vector plus scalar)

These instructions are under SVE Memory - Non-temporal and Multi-register Store.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	msz		0	0	Rm					0	0	1	Pg			Zn					Zt				

Decode fields msz	Instruction Details
00	STNT1B (vector plus scalar)
01	STNT1H (vector plus scalar)
10	STNT1W (vector plus scalar)
11	STNT1D (vector plus scalar)

SVE contiguous non-temporal store (scalar plus scalar)

These instructions are under SVE Memory - Non-temporal and Multi-register Store.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	msz		0	0	Rm					0	1	1	Pg			Rn					Zt				

Decode fields msz	Instruction Details
00	STNT1B (scalar plus scalar)
01	STNT1H (scalar plus scalar)
10	STNT1W (scalar plus scalar)
11	STNT1D (scalar plus scalar)

SVE2 32-bit scatter non-temporal store (vector plus scalar)

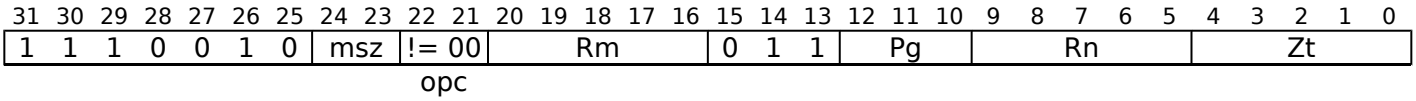
These instructions are under SVE Memory - Non-temporal and Multi-register Store.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	msz		1	0	Rm					0	0	1	Pg			Zn					Zt				

Decode fields msz	Instruction Details
00	STNT1B (vector plus scalar)
01	STNT1H (vector plus scalar)
10	STNT1W (vector plus scalar)
11	UNALLOCATED

SVE store multiple structures (scalar plus scalar)

These instructions are under SVE Memory - Non-temporal and Multi-register Store.

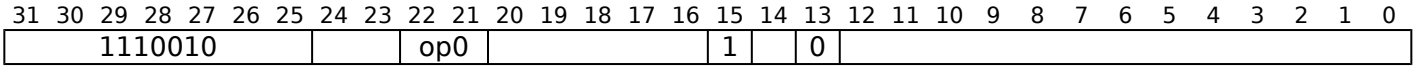


The following constraints also apply to this encoding: opc != 00 && opc != 00

Decode fields msz	opc	Instruction Details
00	01	ST2B (scalar plus scalar)
00	10	ST3B (scalar plus scalar)
00	11	ST4B (scalar plus scalar)
01	01	ST2H (scalar plus scalar)
01	10	ST3H (scalar plus scalar)
01	11	ST4H (scalar plus scalar)
10	01	ST2W (scalar plus scalar)
10	10	ST3W (scalar plus scalar)
10	11	ST4W (scalar plus scalar)
11	01	ST2D (scalar plus scalar)
11	10	ST3D (scalar plus scalar)
11	11	ST4D (scalar plus scalar)

SVE Memory - Scatter with Optional Sign Extend

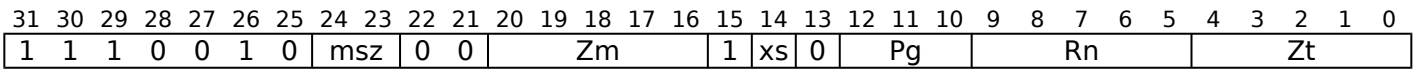
These instructions are under SVE encodings.



Decode fields op0	Instruction details
00	SVE 64-bit scatter store (scalar plus unpacked 32-bit unscaled offsets)
01	SVE 64-bit scatter store (scalar plus unpacked 32-bit scaled offsets)
10	SVE 32-bit scatter store (scalar plus 32-bit unscaled offsets)
11	SVE 32-bit scatter store (scalar plus 32-bit scaled offsets)

SVE 64-bit scatter store (scalar plus unpacked 32-bit unscaled offsets)

These instructions are under SVE Memory - Scatter with Optional Sign Extend.



Decode fields msz	Instruction Details
00	ST1B (scalar plus vector)
01	ST1H (scalar plus vector)
10	ST1W (scalar plus vector)
11	ST1D (scalar plus vector)

SVE 64-bit scatter store (scalar plus unpacked 32-bit scaled offsets)

These instructions are under SVE Memory - Scatter with Optional Sign Extend.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	msz		0	1	Zm					1	xs	0	Pg			Rn					Zt				

Decode fields msz	Instruction Details
00	UNALLOCATED
01	ST1H (scalar plus vector)
10	ST1W (scalar plus vector)
11	ST1D (scalar plus vector)

SVE 32-bit scatter store (scalar plus 32-bit unscaled offsets)

These instructions are under SVE Memory - Scatter with Optional Sign Extend.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	msz		1	0	Zm					1	xs	0	Pg			Rn					Zt				

Decode fields msz	Instruction Details
00	ST1B (scalar plus vector)
01	ST1H (scalar plus vector)
10	ST1W (scalar plus vector)
11	UNALLOCATED

SVE 32-bit scatter store (scalar plus 32-bit scaled offsets)

These instructions are under SVE Memory - Scatter with Optional Sign Extend.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	msz		1	1	Zm					1	xs	0	Pg			Rn					Zt				

Decode fields msz	Instruction Details
00	UNALLOCATED
01	ST1H (scalar plus vector)
10	ST1W (scalar plus vector)
11	UNALLOCATED

SVE Memory - Scatter

These instructions are under SVE encodings.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1110010									op0							101															

Decode fields op0	Instruction details
00	SVE 64-bit scatter store (scalar plus 64-bit unscaled offsets)
01	SVE 64-bit scatter store (scalar plus 64-bit scaled offsets)
10	SVE 64-bit scatter store (vector plus immediate)
11	SVE 32-bit scatter store (vector plus immediate)

SVE 64-bit scatter store (scalar plus 64-bit unscaled offsets)

These instructions are under SVE Memory - Scatter.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	msz		0	0	Zm					1	0	1	Pg			Rn					Zt				

Decode fields msz	Instruction Details
00	ST1B (scalar plus vector)
01	ST1H (scalar plus vector)
10	ST1W (scalar plus vector)
11	ST1D (scalar plus vector)

SVE 64-bit scatter store (scalar plus 64-bit scaled offsets)

These instructions are under SVE Memory - Scatter.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	msz		0	1	Zm					1	0	1	Pg			Rn					Zt				

Decode fields msz	Instruction Details
00	UNALLOCATED
01	ST1H (scalar plus vector)
10	ST1W (scalar plus vector)
11	ST1D (scalar plus vector)

SVE 64-bit scatter store (vector plus immediate)

These instructions are under SVE Memory - Scatter.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	msz		1	0	imm5					1	0	1	Pg			Zn					Zt				

Decode fields msz	Instruction Details
00	ST1B (vector plus immediate)
01	ST1H (vector plus immediate)
10	ST1W (vector plus immediate)
11	ST1D (vector plus immediate)

SVE 32-bit scatter store (vector plus immediate)

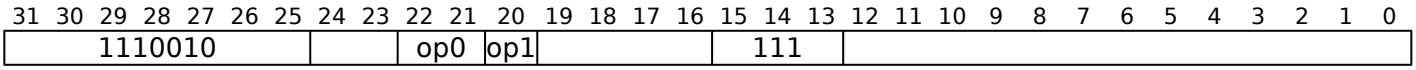
These instructions are under SVE Memory - Scatter.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	0	msz		1	1	imm5					1	0	1	Pg			Zn					Zt				

Decode fields msz	Instruction Details
00	ST1B (vector plus immediate)
01	ST1H (vector plus immediate)
10	ST1W (vector plus immediate)
11	UNALLOCATED

SVE Memory - Contiguous Store with Immediate Offset

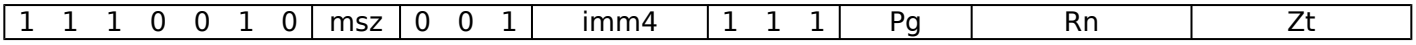
These instructions are under SVE encodings.



Decode fields op0	Decode fields op1	Instruction details
00	1	SVE contiguous non-temporal store (scalar plus immediate)
!= 00	1	SVE store multiple structures (scalar plus immediate)
	0	SVE contiguous store (scalar plus immediate)

SVE contiguous non-temporal store (scalar plus immediate)

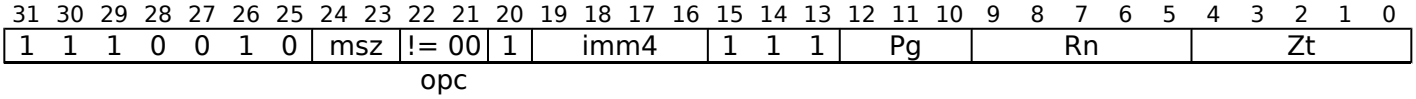
These instructions are under SVE Memory - Contiguous Store with Immediate Offset.



Decode fields msz	Instruction Details
00	STNT1B (scalar plus immediate)
01	STNT1H (scalar plus immediate)
10	STNT1W (scalar plus immediate)
11	STNT1D (scalar plus immediate)

SVE store multiple structures (scalar plus immediate)

These instructions are under SVE Memory - Contiguous Store with Immediate Offset.

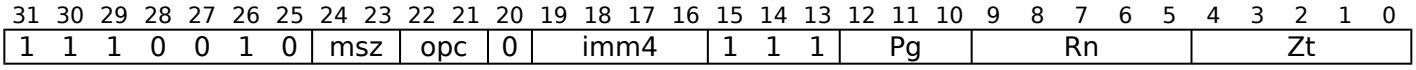


The following constraints also apply to this encoding: opc != 00 && opc != 00

Decode fields msz	Decode fields opc	Instruction Details
00	01	ST2B (scalar plus immediate)
00	10	ST3B (scalar plus immediate)
00	11	ST4B (scalar plus immediate)
01	01	ST2H (scalar plus immediate)
01	10	ST3H (scalar plus immediate)
01	11	ST4H (scalar plus immediate)
10	01	ST2W (scalar plus immediate)
10	10	ST3W (scalar plus immediate)
10	11	ST4W (scalar plus immediate)
11	01	ST2D (scalar plus immediate)
11	10	ST3D (scalar plus immediate)
11	11	ST4D (scalar plus immediate)

SVE contiguous store (scalar plus immediate)

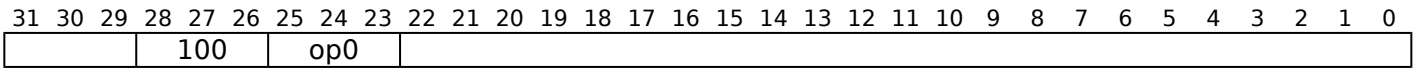
These instructions are under SVE Memory - Contiguous Store with Immediate Offset.



Decode fields msz	opc	Instruction Details
00		ST1B (scalar plus immediate)
01		ST1H (scalar plus immediate)
1x	0x	UNALLOCATED
10	1x	ST1W (scalar plus immediate)
11	10	UNALLOCATED
11	11	ST1D (scalar plus immediate)

Data Processing -- Immediate

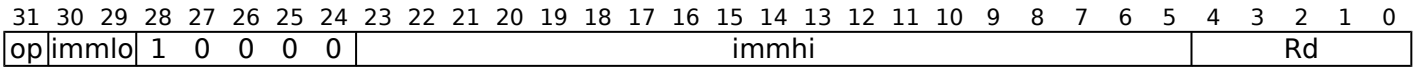
These instructions are under the top-level.



Decode fields op0	Instruction details
00x	PC-rel. addressing
010	Add/subtract (immediate)
011	Add/subtract (immediate, with tags)
100	Logical (immediate)
101	Move wide (immediate)
110	Bitfield
111	Extract

PC-rel. addressing

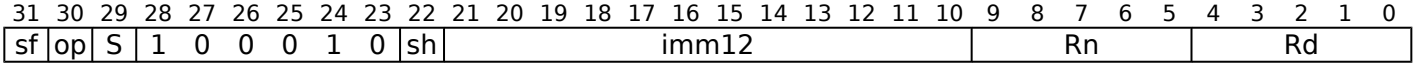
These instructions are under Data Processing -- Immediate.



Decode fields op	Instruction Details
0	ADR
1	ADRP

Add/subtract (immediate)

These instructions are under Data Processing -- Immediate.

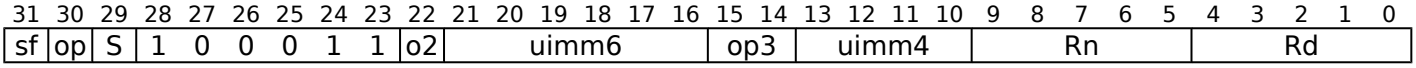


Decode fields sf	op	S	Instruction Details
0	0	0	ADD (immediate) — 32-bit
0	0	1	ADDS (immediate) — 32-bit
0	1	0	SUB (immediate) — 32-bit
0	1	1	SUBS (immediate) — 32-bit
1	0	0	ADD (immediate) — 64-bit

Decode fields sf	op	S	Instruction Details
1	0	1	ADDS (immediate) — 64-bit
1	1	0	SUB (immediate) — 64-bit
1	1	1	SUBS (immediate) — 64-bit

Add/subtract (immediate, with tags)

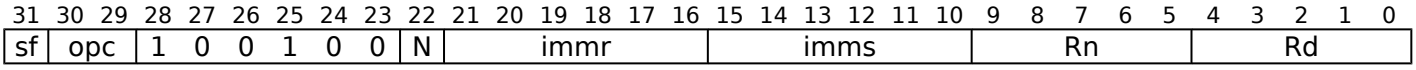
These instructions are under Data Processing -- Immediate.



Decode fields sf	op	S	o2	Instruction Details	Feature
			1	UNALLOCATED	-
0			0	UNALLOCATED	-
1		1	0	UNALLOCATED	-
1	0	0	0	ADDG	FEAT_MTE
1	1	0	0	SUBG	FEAT_MTE

Logical (immediate)

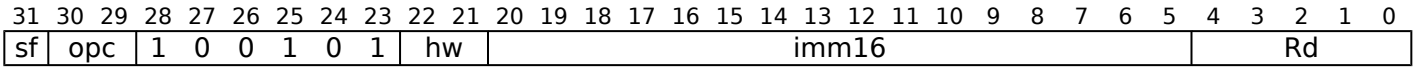
These instructions are under Data Processing -- Immediate.



Decode fields sf	opc	N	Instruction Details
0		1	UNALLOCATED
0	00	0	AND (immediate) — 32-bit
0	01	0	ORR (immediate) — 32-bit
0	10	0	EOR (immediate) — 32-bit
0	11	0	ANDS (immediate) — 32-bit
1	00		AND (immediate) — 64-bit
1	01		ORR (immediate) — 64-bit
1	10		EOR (immediate) — 64-bit
1	11		ANDS (immediate) — 64-bit

Move wide (immediate)

These instructions are under Data Processing -- Immediate.

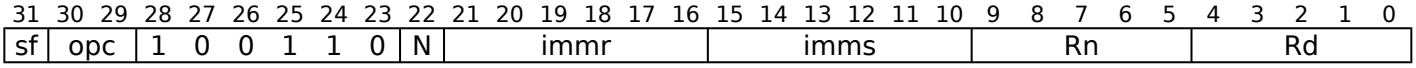


Decode fields sf	opc	hw	Instruction Details
	01		UNALLOCATED
0		1x	UNALLOCATED
0	00	0x	MOVN — 32-bit
0	10	0x	MOVZ — 32-bit
0	11	0x	MOVK — 32-bit

Decode fields sf	opc	hw	Instruction Details
1	00		MOVN — 64-bit
1	10		MOVZ — 64-bit
1	11		MOVK — 64-bit

Bitfield

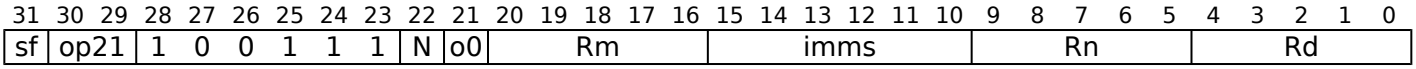
These instructions are under Data Processing -- Immediate.



Decode fields sf	opc	N	Instruction Details
	11		UNALLOCATED
0		1	UNALLOCATED
0	00	0	SBFM — 32-bit
0	01	0	BFM — 32-bit
0	10	0	UBFM — 32-bit
1		0	UNALLOCATED
1	00	1	SBFM — 64-bit
1	01	1	BFM — 64-bit
1	10	1	UBFM — 64-bit

Extract

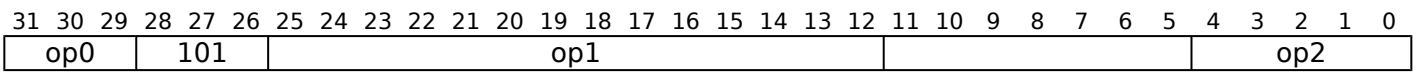
These instructions are under Data Processing -- Immediate.



Decode fields sf	op21	N	o0	imms	Instruction Details
	x1				UNALLOCATED
	00		1		UNALLOCATED
	1x				UNALLOCATED
0				1xxxxx	UNALLOCATED
0		1			UNALLOCATED
0	00	0	0	0xxxxx	EXTR — 32-bit
1		0			UNALLOCATED
1	00	1	0		EXTR — 64-bit

Branches, Exception Generating and System instructions

These instructions are under the top-level.

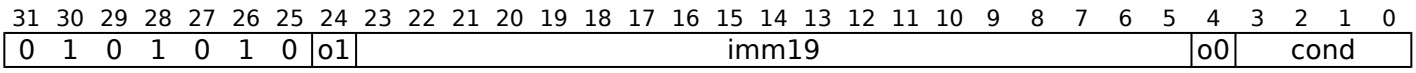


Decode fields op0	op1	op2	Instruction details
010	0xxxxxxxxxxxx		Conditional branch (immediate)
110	00xxxxxxxxxxx		Exception generation

110	01000000110001		System instructions with register argument
110	01000000110010	11111	Hints
110	01000000110011		Barriers
110	0100000xxx0100		PSTATE
110	0100100xxxxxxx		System with result
110	0100x01xxxxxxx		System instructions
110	0100x1xxxxxxxx		System register move
110	1xxxxxxxxxxxxx		Unconditional branch (register)
x00			Unconditional branch (immediate)
x01	0xxxxxxxxxxxxx		Compare and branch (immediate)
x01	1xxxxxxxxxxxxx		Test and branch (immediate)

Conditional branch (immediate)

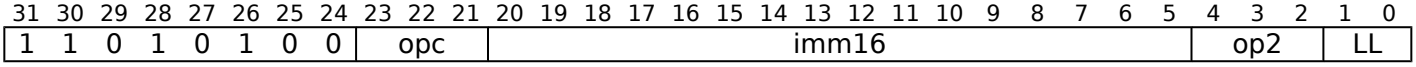
These instructions are under Branches, Exception Generating and System instructions.



Decode fields o1	o0	Instruction Details	Feature
0	0	B.cond	-
0	1	BC.cond	FEAT_HBC
1		UNALLOCATED	-

Exception generation

These instructions are under Branches, Exception Generating and System instructions.



Decode fields opc	op2	LL	Instruction Details	Feature
	001		UNALLOCATED	-
	01x		UNALLOCATED	-
	1xx		UNALLOCATED	-
000	000	00	UNALLOCATED	-
000	000	01	SVC	-
000	000	10	HVC	-
000	000	11	SMC	-
001	000	x1	UNALLOCATED	-
001	000	00	BRK	-
001	000	1x	UNALLOCATED	-
010	000	x1	UNALLOCATED	-
010	000	00	HLT	-
010	000	1x	UNALLOCATED	-
011	000	00	TCANCEL	FEAT_TME
011	000	01	UNALLOCATED	-
011	000	1x	UNALLOCATED	-
100	000		UNALLOCATED	-
101	000	00	UNALLOCATED	-

Decode fields opc	op2	LL	Instruction Details	Feature
101	000	01	DCPS1	-
101	000	10	DCPS2	-
101	000	11	DCPS3	-
110	000		UNALLOCATED	-
111	000		UNALLOCATED	-

System instructions with register argument

These instructions are under Branches, Exception Generating and System instructions.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	0	1	CRm				op2			Rt				

Decode fields CRm	op2	Instruction Details	Feature
!= 0000		UNALLOCATED	-
0000	000	WFET	FEAT_WFxT
0000	001	WFIT	FEAT_WFxT
0000	01x	UNALLOCATED	-
0000	1xx	UNALLOCATED	-

Hints

These instructions are under Branches, Exception Generating and System instructions.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	CRm				op2			1	1	1	1	1

Decode fields CRm	op2	Instruction Details	Feature
		HINT	-
0000	000	NOP	-
0000	001	YIELD	-
0000	010	WFE	-
0000	011	WFI	-
0000	100	SEV	-
0000	101	SEVL	-
0000	110	DGH	FEAT_DGH
0000	111	XPACD, XPACI, XPACLRI	FEAT_PAuth
0001	000	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIA1716	FEAT_PAuth
0001	010	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIB1716	FEAT_PAuth
0001	100	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIA1716	FEAT_PAuth
0001	110	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIB1716	FEAT_PAuth
0010	000	ESB	FEAT_RAS
0010	001	PSB CSYNC	FEAT_SPE
0010	010	TSB CSYNC	FEAT_TRF
0010	100	CSDB	-
0011	000	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIAZ	FEAT_PAuth
0011	001	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIASP	FEAT_PAuth
0011	010	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIBZ	FEAT_PAuth
0011	011	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIBSP	FEAT_PAuth

Decode fields CRm	op2	Instruction Details	Feature
0011	100	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIAZ	FEAT_PAuth
0011	101	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIASP	FEAT_PAuth
0011	110	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIBZ	FEAT_PAuth
0011	111	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIBSP	FEAT_PAuth
0100	xx0	BTI	FEAT_BTI

Barriers

These instructions are under Branches, Exception Generating and System instructions.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	CRm				op2			Rt				

CRm	Decode fields op2	Rt	Instruction Details	Feature
	000		UNALLOCATED	-
	001	!= 11111	UNALLOCATED	-
	010	11111	CLREX	-
	100	11111	DSB — memory barrier	-
	101	11111	DMB	-
	110	11111	ISB	-
	111	!= 11111	UNALLOCATED	-
	111	11111	SB	FEAT_SB
xx0x	001	11111	UNALLOCATED	-
xx10	001	11111	DSB — Memory nXS barrier	FEAT_XS
xx11	001	11111	UNALLOCATED	-
0000	011	11111	TCOMMIT	FEAT_TME
0001	011		UNALLOCATED	-
001x	011		UNALLOCATED	-
01xx	011		UNALLOCATED	-
1xxx	011		UNALLOCATED	-

PSTATE

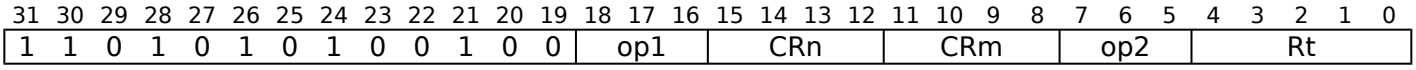
These instructions are under Branches, Exception Generating and System instructions.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	0	1	0	1	0	0	0	0	0	op1			0	1	0	0	CRm				op2			Rt				

op1	Decode fields op2	Rt	Instruction Details	Feature
		!= 11111	UNALLOCATED	-
		11111	MSR (immediate)	-
000	000	11111	CFINV	FEAT_FlagM
000	001	11111	XAFLAG	FEAT_FlagM2
000	010	11111	AXFLAG	FEAT_FlagM2

System with result

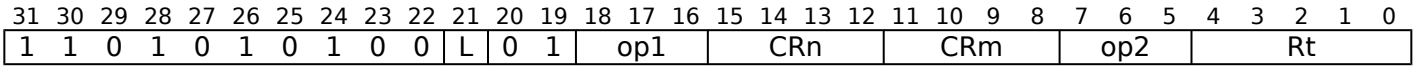
These instructions are under Branches, Exception Generating and System instructions.



Decode fields				Instruction Details	Feature
op1	CRn	CRm	op2		
!= 011				UNALLOCATED	-
011	!= 0011			UNALLOCATED	-
011	0011		!= 011	UNALLOCATED	-
011	0011	!= 000x	011	UNALLOCATED	-
011	0011	0000	011	TSTART	FEAT_TME
011	0011	0001	011	TTEST	FEAT_TME

System instructions

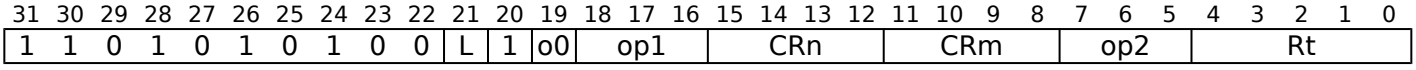
These instructions are under Branches, Exception Generating and System instructions.



Decode fields	Instruction Details
L	
0	SYS
1	SYSL

System register move

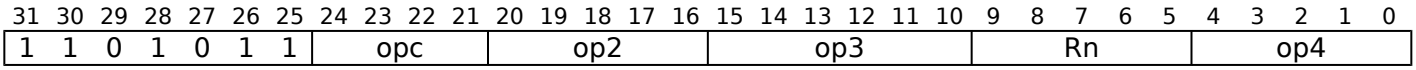
These instructions are under Branches, Exception Generating and System instructions.



Decode fields	Instruction Details
L	
0	MSR (register)
1	MRS

Unconditional branch (register)

These instructions are under Branches, Exception Generating and System instructions.



Decode fields					Instruction Details	Feature
opc	op2	op3	Rn	op4		
	!= 11111				UNALLOCATED	-
0000	11111	000000		!= 00000	UNALLOCATED	-
0000	11111	000000		00000	BR	-
0000	11111	000001			UNALLOCATED	-
0000	11111	000010		!= 11111	UNALLOCATED	-
0000	11111	000010		11111	BRAA, BRAAZ, BRAB, BRABZ — key A, zero modifier	FEAT_PAuth

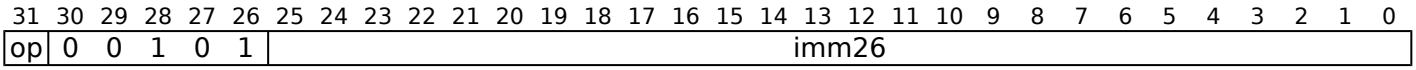
Decode fields						
opc	op2	op3	Rn	op4	Instruction Details	Feature
0000	11111	000011		!= 11111	UNALLOCATED	-
0000	11111	000011		11111	BRAA, BRAAZ, BRAB, BRABZ — key B, zero modifier	FEAT_PAuth
0000	11111	0001xx			UNALLOCATED	-
0000	11111	001xxx			UNALLOCATED	-
0000	11111	01xxxx			UNALLOCATED	-
0000	11111	1xxxxx			UNALLOCATED	-
0001	11111	000000		!= 00000	UNALLOCATED	-
0001	11111	000000		00000	BLR	-
0001	11111	000001			UNALLOCATED	-
0001	11111	000010		!= 11111	UNALLOCATED	-
0001	11111	000010		11111	BLRAA, BLRAAZ, BLRAB, BLRABZ — key A, zero modifier	FEAT_PAuth
0001	11111	000011		!= 11111	UNALLOCATED	-
0001	11111	000011		11111	BLRAA, BLRAAZ, BLRAB, BLRABZ — key B, zero modifier	FEAT_PAuth
0001	11111	0001xx			UNALLOCATED	-
0001	11111	001xxx			UNALLOCATED	-
0001	11111	01xxxx			UNALLOCATED	-
0001	11111	1xxxxx			UNALLOCATED	-
0010	11111	000000		!= 00000	UNALLOCATED	-
0010	11111	000000		00000	RET	-
0010	11111	000001			UNALLOCATED	-
0010	11111	000010	!= 11111	!= 11111	UNALLOCATED	-
0010	11111	000010	!= 11111	11111	UNALLOCATED	-
0010	11111	000010	11111	!= 11111	UNALLOCATED	-
0010	11111	000010	11111	11111	RETAA, RETAB — RETAA	FEAT_PAuth
0010	11111	000011	!= 11111	!= 11111	UNALLOCATED	-
0010	11111	000011	!= 11111	11111	UNALLOCATED	-
0010	11111	000011	11111	!= 11111	UNALLOCATED	-
0010	11111	000011	11111	11111	RETAA, RETAB — RETAB	FEAT_PAuth
0010	11111	0001xx			UNALLOCATED	-
0010	11111	001xxx			UNALLOCATED	-
0010	11111	01xxxx			UNALLOCATED	-
0010	11111	1xxxxx			UNALLOCATED	-
0011	11111				UNALLOCATED	-
0100	11111	000000	!= 11111	!= 00000	UNALLOCATED	-

Decode fields						
opc	op2	op3	Rn	op4	Instruction Details	Feature
0100	11111	000000	!= 11111	00000	UNALLOCATED	-
0100	11111	000000	11111	!= 00000	UNALLOCATED	-
0100	11111	000000	11111	00000	ERET	-
0100	11111	000001			UNALLOCATED	-
0100	11111	000010	!= 11111	!= 11111	UNALLOCATED	-
0100	11111	000010	!= 11111	11111	UNALLOCATED	-
0100	11111	000010	11111	!= 11111	UNALLOCATED	-
0100	11111	000010	11111	11111	ERETAA, ERETAB — ERETAA	FEAT_PAuth
0100	11111	000011	!= 11111	!= 11111	UNALLOCATED	-
0100	11111	000011	!= 11111	11111	UNALLOCATED	-
0100	11111	000011	11111	!= 11111	UNALLOCATED	-
0100	11111	000011	11111	11111	ERETAA, ERETAB — ERETAB	FEAT_PAuth
0100	11111	0001xx			UNALLOCATED	-
0100	11111	001xxx			UNALLOCATED	-
0100	11111	01xxxx			UNALLOCATED	-
0100	11111	1xxxxx			UNALLOCATED	-
0101	11111	!= 000000			UNALLOCATED	-
0101	11111	000000	!= 11111	!= 00000	UNALLOCATED	-
0101	11111	000000	!= 11111	00000	UNALLOCATED	-
0101	11111	000000	11111	!= 00000	UNALLOCATED	-
0101	11111	000000	11111	00000	DRPS	-
011x	11111				UNALLOCATED	-
1000	11111	00000x			UNALLOCATED	-
1000	11111	000010			BRAA, BRAAZ, BRAB, BRABZ — key A, register modifier	FEAT_PAuth
1000	11111	000011			BRAA, BRAAZ, BRAB, BRABZ — key B, register modifier	FEAT_PAuth
1000	11111	0001xx			UNALLOCATED	-
1000	11111	001xxx			UNALLOCATED	-
1000	11111	01xxxx			UNALLOCATED	-
1000	11111	1xxxxx			UNALLOCATED	-
1001	11111	00000x			UNALLOCATED	-
1001	11111	000010			BLRAA, BLRAAZ, BLRAB, BLRABZ — key A, register modifier	FEAT_PAuth
1001	11111	000011			BLRAA, BLRAAZ, BLRAB, BLRABZ — key B, register modifier	FEAT_PAuth
1001	11111	0001xx			UNALLOCATED	-

Decode fields						
opc	op2	op3	Rn	op4	Instruction Details	Feature
1001	11111	001xxx			UNALLOCATED	-
1001	11111	01xxxx			UNALLOCATED	-
1001	11111	1xxxxx			UNALLOCATED	-
101x	11111				UNALLOCATED	-
11xx	11111				UNALLOCATED	-

Unconditional branch (immediate)

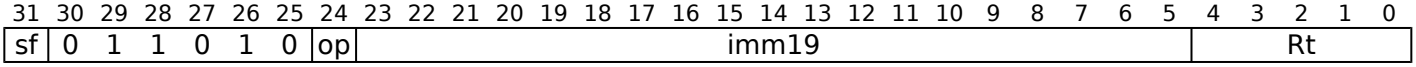
These instructions are under Branches, Exception Generating and System instructions.



Decode fields	
op	Instruction Details
0	B
1	BL

Compare and branch (immediate)

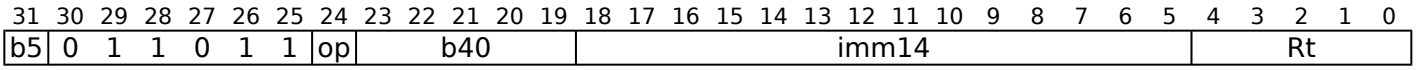
These instructions are under Branches, Exception Generating and System instructions.



Decode fields		
sf	op	Instruction Details
0	0	CBZ — 32-bit
0	1	CBNZ — 32-bit
1	0	CBZ — 64-bit
1	1	CBNZ — 64-bit

Test and branch (immediate)

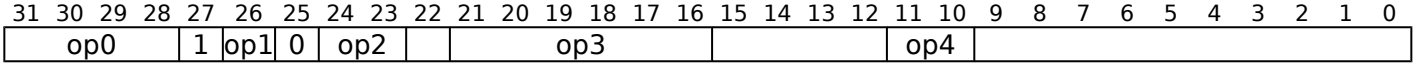
These instructions are under Branches, Exception Generating and System instructions.



Decode fields	
op	Instruction Details
0	TBZ
1	TBNZ

Loads and Stores

These instructions are under the top-level.



Decode fields					
op0	op1	op2	op3	op4	Instruction details

0x00	0	00	1xxxxx		Compare and swap pair
0x00	1	00	000000		Advanced SIMD load/store multiple structures
0x00	1	01	0xxxxx		Advanced SIMD load/store multiple structures (post-indexed)
0x00	1	0x	1xxxxx		UNALLOCATED
0x00	1	10	x00000		Advanced SIMD load/store single structure
0x00	1	11			Advanced SIMD load/store single structure (post-indexed)
0x00	1	x0	x1xxxx		UNALLOCATED
0x00	1	x0	xx1xxx		UNALLOCATED
0x00	1	x0	xxx1xx		UNALLOCATED
0x00	1	x0	xxxx1x		UNALLOCATED
0x00	1	x0	xxxxx1		UNALLOCATED
1101	0	1x	1xxxxx		Load/store memory tags
1x00	0	00	1xxxxx		Load/store exclusive pair
1x00	1				UNALLOCATED
xx00	0	00	0xxxxx		Load/store exclusive register
xx00	0	01	0xxxxx		Load/store ordered
xx00	0	01	1xxxxx		Compare and swap
xx01	0	1x	0xxxxx	00	LDAPR/STLR (unscaled immediate)
xx01		0x			Load register (literal)
xx01		1x	0xxxxx	01	Memory Copy and Memory Set
xx10		00			Load/store no-allocate pair (offset)
xx10		01			Load/store register pair (post-indexed)
xx10		10			Load/store register pair (offset)
xx10		11			Load/store register pair (pre-indexed)
xx11		0x	0xxxxx	00	Load/store register (unscaled immediate)
xx11		0x	0xxxxx	01	Load/store register (immediate post-indexed)
xx11		0x	0xxxxx	10	Load/store register (unprivileged)
xx11		0x	0xxxxx	11	Load/store register (immediate pre-indexed)
xx11		0x	1xxxxx	00	Atomic memory operations
xx11		0x	1xxxxx	10	Load/store register (register offset)
xx11		0x	1xxxxx	x1	Load/store register (pac)
xx11		1x			Load/store register (unsigned immediate)

Compare and swap pair

These instructions are under Loads and Stores.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	sz	0	0	1	0	0	0	0	L	1	Rs					o0	Rt2					Rn					Rt				

sz	L	o0	Rt2	Instruction Details	Feature
			!= 11111	UNALLOCATED	-
0	0	0	11111	CASP, CASPA, CASPAL, CASPL — 32-bit CASP	FEAT_LSE
0	0	1	11111	CASP, CASPA, CASPAL, CASPL — 32-bit CASPL	FEAT_LSE
0	1	0	11111	CASP, CASPA, CASPAL, CASPL — 32-bit CASPA	FEAT_LSE
0	1	1	11111	CASP, CASPA, CASPAL, CASPL — 32-bit CASPAL	FEAT_LSE
1	0	0	11111	CASP, CASPA, CASPAL, CASPL — 64-bit CASP	FEAT_LSE
1	0	1	11111	CASP, CASPA, CASPAL, CASPL — 64-bit CASPL	FEAT_LSE
1	1	0	11111	CASP, CASPA, CASPAL, CASPL — 64-bit CASPA	FEAT_LSE

Decode fields				Instruction Details	Feature
sz	L	o0	Rt2		
1	1	1	11111	CASP, CASPA, CASPAL, CASPL — 64-bit CASPAL	FEAT_LSE

Advanced SIMD load/store multiple structures

These instructions are under Loads and Stores.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	0	L	0	0	0	0	0	0	opcode				size		Rn					Rt				

Decode fields		Instruction Details
L	opcode	
0	0000	ST4 (multiple structures)
0	0001	UNALLOCATED
0	0010	ST1 (multiple structures) — four registers
0	0011	UNALLOCATED
0	0100	ST3 (multiple structures)
0	0101	UNALLOCATED
0	0110	ST1 (multiple structures) — three registers
0	0111	ST1 (multiple structures) — one register
0	1000	ST2 (multiple structures)
0	1001	UNALLOCATED
0	1010	ST1 (multiple structures) — two registers
0	1011	UNALLOCATED
0	11xx	UNALLOCATED
1	0000	LD4 (multiple structures)
1	0001	UNALLOCATED
1	0010	LD1 (multiple structures) — four registers
1	0011	UNALLOCATED
1	0100	LD3 (multiple structures)
1	0101	UNALLOCATED
1	0110	LD1 (multiple structures) — three registers
1	0111	LD1 (multiple structures) — one register
1	1000	LD2 (multiple structures)
1	1001	UNALLOCATED
1	1010	LD1 (multiple structures) — two registers
1	1011	UNALLOCATED
1	11xx	UNALLOCATED

Advanced SIMD load/store multiple structures (post-indexed)

These instructions are under Loads and Stores.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	0	1	L	0	Rm					opcode				size		Rn					Rt				

Decode fields			Instruction Details
L	Rm	opcode	
0		0001	UNALLOCATED
0		0011	UNALLOCATED
0		0101	UNALLOCATED

Decode fields			Instruction Details
L	Rm	opcode	
0		1001	UNALLOCATED
0		1011	UNALLOCATED
0		11xx	UNALLOCATED
0	!= 11111	0000	ST4 (multiple structures) — register offset
0	!= 11111	0010	ST1 (multiple structures) — four registers, register offset
0	!= 11111	0100	ST3 (multiple structures) — register offset
0	!= 11111	0110	ST1 (multiple structures) — three registers, register offset
0	!= 11111	0111	ST1 (multiple structures) — one register, register offset
0	!= 11111	1000	ST2 (multiple structures) — register offset
0	!= 11111	1010	ST1 (multiple structures) — two registers, register offset
0	11111	0000	ST4 (multiple structures) — immediate offset
0	11111	0010	ST1 (multiple structures) — four registers, immediate offset
0	11111	0100	ST3 (multiple structures) — immediate offset
0	11111	0110	ST1 (multiple structures) — three registers, immediate offset
0	11111	0111	ST1 (multiple structures) — one register, immediate offset
0	11111	1000	ST2 (multiple structures) — immediate offset
0	11111	1010	ST1 (multiple structures) — two registers, immediate offset
1		0001	UNALLOCATED
1		0011	UNALLOCATED
1		0101	UNALLOCATED
1		1001	UNALLOCATED
1		1011	UNALLOCATED
1		11xx	UNALLOCATED
1	!= 11111	0000	LD4 (multiple structures) — register offset
1	!= 11111	0010	LD1 (multiple structures) — four registers, register offset
1	!= 11111	0100	LD3 (multiple structures) — register offset
1	!= 11111	0110	LD1 (multiple structures) — three registers, register offset
1	!= 11111	0111	LD1 (multiple structures) — one register, register offset
1	!= 11111	1000	LD2 (multiple structures) — register offset
1	!= 11111	1010	LD1 (multiple structures) — two registers, register offset
1	11111	0000	LD4 (multiple structures) — immediate offset
1	11111	0010	LD1 (multiple structures) — four registers, immediate offset
1	11111	0100	LD3 (multiple structures) — immediate offset
1	11111	0110	LD1 (multiple structures) — three registers, immediate offset
1	11111	0111	LD1 (multiple structures) — one register, immediate offset
1	11111	1000	LD2 (multiple structures) — immediate offset
1	11111	1010	LD1 (multiple structures) — two registers, immediate offset

Advanced SIMD load/store single structure

These instructions are under Loads and Stores.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	0	L	R	0	0	0	0	0	opcode			S	size		Rn					Rt				

Decode fields					Instruction Details
L	R	opcode	S	size	
0		11x			UNALLOCATED
0	0	000			ST1 (single structure) — 8-bit

Decode fields					Instruction Details
L	R	opcode	S	size	
0	0	001			ST3 (single structure) — 8-bit
0	0	010		x0	ST1 (single structure) — 16-bit
0	0	010		x1	UNALLOCATED
0	0	011		x0	ST3 (single structure) — 16-bit
0	0	011		x1	UNALLOCATED
0	0	100		00	ST1 (single structure) — 32-bit
0	0	100		1x	UNALLOCATED
0	0	100	0	01	ST1 (single structure) — 64-bit
0	0	100	1	01	UNALLOCATED
0	0	101		00	ST3 (single structure) — 32-bit
0	0	101		10	UNALLOCATED
0	0	101	0	01	ST3 (single structure) — 64-bit
0	0	101	0	11	UNALLOCATED
0	0	101	1	x1	UNALLOCATED
0	1	000			ST2 (single structure) — 8-bit
0	1	001			ST4 (single structure) — 8-bit
0	1	010		x0	ST2 (single structure) — 16-bit
0	1	010		x1	UNALLOCATED
0	1	011		x0	ST4 (single structure) — 16-bit
0	1	011		x1	UNALLOCATED
0	1	100		00	ST2 (single structure) — 32-bit
0	1	100		10	UNALLOCATED
0	1	100	0	01	ST2 (single structure) — 64-bit
0	1	100	0	11	UNALLOCATED
0	1	100	1	x1	UNALLOCATED
0	1	101		00	ST4 (single structure) — 32-bit
0	1	101		10	UNALLOCATED
0	1	101	0	01	ST4 (single structure) — 64-bit
0	1	101	0	11	UNALLOCATED
0	1	101	1	x1	UNALLOCATED
1	0	000			LD1 (single structure) — 8-bit
1	0	001			LD3 (single structure) — 8-bit
1	0	010		x0	LD1 (single structure) — 16-bit
1	0	010		x1	UNALLOCATED
1	0	011		x0	LD3 (single structure) — 16-bit
1	0	011		x1	UNALLOCATED
1	0	100		00	LD1 (single structure) — 32-bit
1	0	100		1x	UNALLOCATED
1	0	100	0	01	LD1 (single structure) — 64-bit
1	0	100	1	01	UNALLOCATED
1	0	101		00	LD3 (single structure) — 32-bit
1	0	101		10	UNALLOCATED
1	0	101	0	01	LD3 (single structure) — 64-bit
1	0	101	0	11	UNALLOCATED
1	0	101	1	x1	UNALLOCATED
1	0	110	0		LD1R
1	0	110	1		UNALLOCATED

Decode fields					Instruction Details
L	R	opcode	S	size	
1	0	111	0		LD3R
1	0	111	1		UNALLOCATED
1	1	000			LD2 (single structure) — 8-bit
1	1	001			LD4 (single structure) — 8-bit
1	1	010		x0	LD2 (single structure) — 16-bit
1	1	010		x1	UNALLOCATED
1	1	011		x0	LD4 (single structure) — 16-bit
1	1	011		x1	UNALLOCATED
1	1	100		00	LD2 (single structure) — 32-bit
1	1	100		10	UNALLOCATED
1	1	100	0	01	LD2 (single structure) — 64-bit
1	1	100	0	11	UNALLOCATED
1	1	100	1	x1	UNALLOCATED
1	1	101		00	LD4 (single structure) — 32-bit
1	1	101		10	UNALLOCATED
1	1	101	0	01	LD4 (single structure) — 64-bit
1	1	101	0	11	UNALLOCATED
1	1	101	1	x1	UNALLOCATED
1	1	110	0		LD2R
1	1	110	1		UNALLOCATED
1	1	111	0		LD4R
1	1	111	1		UNALLOCATED

Advanced SIMD load/store single structure (post-indexed)

These instructions are under Loads and Stores.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	0	1	1	L	R	Rm					opcode			S	size		Rn					Rt				

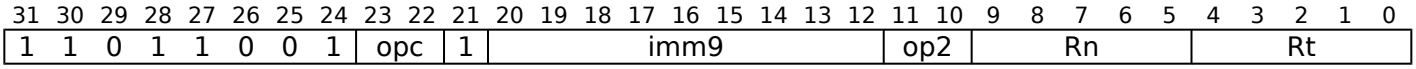
Decode fields						Instruction Details
L	R	Rm	opcode	S	size	
0			11x			UNALLOCATED
0	0		010		x1	UNALLOCATED
0	0		011		x1	UNALLOCATED
0	0		100		1x	UNALLOCATED
0	0		100	1	01	UNALLOCATED
0	0		101		10	UNALLOCATED
0	0		101	0	11	UNALLOCATED
0	0		101	1	x1	UNALLOCATED
0	0	!= 11111	000			ST1 (single structure) — 8-bit, register offset
0	0	!= 11111	001			ST3 (single structure) — 8-bit, register offset
0	0	!= 11111	010		x0	ST1 (single structure) — 16-bit, register offset
0	0	!= 11111	011		x0	ST3 (single structure) — 16-bit, register offset
0	0	!= 11111	100		00	ST1 (single structure) — 32-bit, register offset
0	0	!= 11111	100	0	01	ST1 (single structure) — 64-bit, register offset
0	0	!= 11111	101		00	ST3 (single structure) — 32-bit, register offset
0	0	!= 11111	101	0	01	ST3 (single structure) — 64-bit, register offset
0	0	11111	000			ST1 (single structure) — 8-bit, immediate offset

Decode fields						Instruction Details
L	R	Rm	opcode	S	size	
0	0	11111	001			ST3 (single structure) — 8-bit, immediate offset
0	0	11111	010		x0	ST1 (single structure) — 16-bit, immediate offset
0	0	11111	011		x0	ST3 (single structure) — 16-bit, immediate offset
0	0	11111	100		00	ST1 (single structure) — 32-bit, immediate offset
0	0	11111	100	0	01	ST1 (single structure) — 64-bit, immediate offset
0	0	11111	101		00	ST3 (single structure) — 32-bit, immediate offset
0	0	11111	101	0	01	ST3 (single structure) — 64-bit, immediate offset
0	1		010		x1	UNALLOCATED
0	1		011		x1	UNALLOCATED
0	1		100		10	UNALLOCATED
0	1		100	0	11	UNALLOCATED
0	1		100	1	x1	UNALLOCATED
0	1		101		10	UNALLOCATED
0	1		101	0	11	UNALLOCATED
0	1		101	1	x1	UNALLOCATED
0	1	!= 11111	000			ST2 (single structure) — 8-bit, register offset
0	1	!= 11111	001			ST4 (single structure) — 8-bit, register offset
0	1	!= 11111	010		x0	ST2 (single structure) — 16-bit, register offset
0	1	!= 11111	011		x0	ST4 (single structure) — 16-bit, register offset
0	1	!= 11111	100		00	ST2 (single structure) — 32-bit, register offset
0	1	!= 11111	100	0	01	ST2 (single structure) — 64-bit, register offset
0	1	!= 11111	101		00	ST4 (single structure) — 32-bit, register offset
0	1	!= 11111	101	0	01	ST4 (single structure) — 64-bit, register offset
0	1	11111	000			ST2 (single structure) — 8-bit, immediate offset
0	1	11111	001			ST4 (single structure) — 8-bit, immediate offset
0	1	11111	010		x0	ST2 (single structure) — 16-bit, immediate offset
0	1	11111	011		x0	ST4 (single structure) — 16-bit, immediate offset
0	1	11111	100		00	ST2 (single structure) — 32-bit, immediate offset
0	1	11111	100	0	01	ST2 (single structure) — 64-bit, immediate offset
0	1	11111	101		00	ST4 (single structure) — 32-bit, immediate offset
0	1	11111	101	0	01	ST4 (single structure) — 64-bit, immediate offset
1	0		010		x1	UNALLOCATED
1	0		011		x1	UNALLOCATED
1	0		100		1x	UNALLOCATED
1	0		100	1	01	UNALLOCATED
1	0		101		10	UNALLOCATED
1	0		101	0	11	UNALLOCATED
1	0		101	1	x1	UNALLOCATED
1	0		110	1		UNALLOCATED
1	0		111	1		UNALLOCATED
1	0	!= 11111	000			LD1 (single structure) — 8-bit, register offset
1	0	!= 11111	001			LD3 (single structure) — 8-bit, register offset
1	0	!= 11111	010		x0	LD1 (single structure) — 16-bit, register offset
1	0	!= 11111	011		x0	LD3 (single structure) — 16-bit, register offset
1	0	!= 11111	100		00	LD1 (single structure) — 32-bit, register offset
1	0	!= 11111	100	0	01	LD1 (single structure) — 64-bit, register offset
1	0	!= 11111	101		00	LD3 (single structure) — 32-bit, register offset

Decode fields						Instruction Details
L	R	Rm	opcode	S	size	
1	0	!= 11111	101	0	01	LD3 (single structure) — 64-bit, register offset
1	0	!= 11111	110	0		LD1R — register offset
1	0	!= 11111	111	0		LD3R — register offset
1	0	11111	000			LD1 (single structure) — 8-bit, immediate offset
1	0	11111	001			LD3 (single structure) — 8-bit, immediate offset
1	0	11111	010		x0	LD1 (single structure) — 16-bit, immediate offset
1	0	11111	011		x0	LD3 (single structure) — 16-bit, immediate offset
1	0	11111	100		00	LD1 (single structure) — 32-bit, immediate offset
1	0	11111	100	0	01	LD1 (single structure) — 64-bit, immediate offset
1	0	11111	101		00	LD3 (single structure) — 32-bit, immediate offset
1	0	11111	101	0	01	LD3 (single structure) — 64-bit, immediate offset
1	0	11111	110	0		LD1R — immediate offset
1	0	11111	111	0		LD3R — immediate offset
1	1		010		x1	UNALLOCATED
1	1		011		x1	UNALLOCATED
1	1		100		10	UNALLOCATED
1	1		100	0	11	UNALLOCATED
1	1		100	1	x1	UNALLOCATED
1	1		101		10	UNALLOCATED
1	1		101	0	11	UNALLOCATED
1	1		101	1	x1	UNALLOCATED
1	1		110	1		UNALLOCATED
1	1		111	1		UNALLOCATED
1	1	!= 11111	000			LD2 (single structure) — 8-bit, register offset
1	1	!= 11111	001			LD4 (single structure) — 8-bit, register offset
1	1	!= 11111	010		x0	LD2 (single structure) — 16-bit, register offset
1	1	!= 11111	011		x0	LD4 (single structure) — 16-bit, register offset
1	1	!= 11111	100		00	LD2 (single structure) — 32-bit, register offset
1	1	!= 11111	100	0	01	LD2 (single structure) — 64-bit, register offset
1	1	!= 11111	101		00	LD4 (single structure) — 32-bit, register offset
1	1	!= 11111	101	0	01	LD4 (single structure) — 64-bit, register offset
1	1	!= 11111	110	0		LD2R — register offset
1	1	!= 11111	111	0		LD4R — register offset
1	1	11111	000			LD2 (single structure) — 8-bit, immediate offset
1	1	11111	001			LD4 (single structure) — 8-bit, immediate offset
1	1	11111	010		x0	LD2 (single structure) — 16-bit, immediate offset
1	1	11111	011		x0	LD4 (single structure) — 16-bit, immediate offset
1	1	11111	100		00	LD2 (single structure) — 32-bit, immediate offset
1	1	11111	100	0	01	LD2 (single structure) — 64-bit, immediate offset
1	1	11111	101		00	LD4 (single structure) — 32-bit, immediate offset
1	1	11111	101	0	01	LD4 (single structure) — 64-bit, immediate offset
1	1	11111	110	0		LD2R — immediate offset
1	1	11111	111	0		LD4R — immediate offset

Load/store memory tags

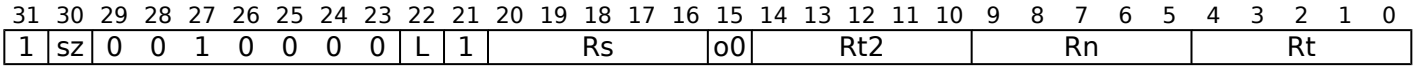
These instructions are under Loads and Stores.



opc	Decode fields imm9	op2	Instruction Details	Feature
00		01	STG — post-index	FEAT_MTE
00		10	STG — signed offset	FEAT_MTE
00		11	STG — pre-index	FEAT_MTE
00	000000000	00	STZGM	FEAT_MTE2
01		00	LDG	FEAT_MTE
01		01	STZG — post-index	FEAT_MTE
01		10	STZG — signed offset	FEAT_MTE
01		11	STZG — pre-index	FEAT_MTE
10		01	ST2G — post-index	FEAT_MTE
10		10	ST2G — signed offset	FEAT_MTE
10		11	ST2G — pre-index	FEAT_MTE
10	!= 000000000	00	UNALLOCATED	-
10	000000000	00	STGM	FEAT_MTE2
11		01	STZ2G — post-index	FEAT_MTE
11		10	STZ2G — signed offset	FEAT_MTE
11		11	STZ2G — pre-index	FEAT_MTE
11	!= 000000000	00	UNALLOCATED	-
11	000000000	00	LDGM	FEAT_MTE2

Load/store exclusive pair

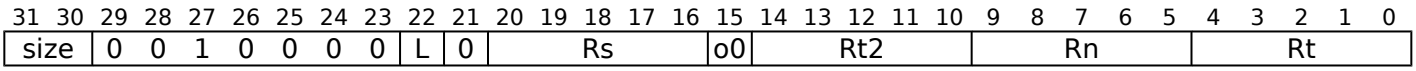
These instructions are under Loads and Stores.



sz	Decode fields L	o0	Instruction Details
0	0	0	STXP — 32-bit
0	0	1	STLXP — 32-bit
0	1	0	LDXP — 32-bit
0	1	1	LDAXP — 32-bit
1	0	0	STXP — 64-bit
1	0	1	STLXP — 64-bit
1	1	0	LDXP — 64-bit
1	1	1	LDAXP — 64-bit

Load/store exclusive register

These instructions are under Loads and Stores.



size	Decode fields L	o0	Instruction Details
00	0	0	STXRB
00	0	1	STLXRB
00	1	0	LDXRB

Decode fields			Instruction Details
size	L	o0	
00	1	1	LDAXRB
01	0	0	STXRH
01	0	1	STLXRH
01	1	0	LDXRH
01	1	1	LDAXRH
10	0	0	STXR — 32-bit
10	0	1	STLXR — 32-bit
10	1	0	LDXR — 32-bit
10	1	1	LDAXR — 32-bit
11	0	0	STXR — 64-bit
11	0	1	STLXR — 64-bit
11	1	0	LDXR — 64-bit
11	1	1	LDAXR — 64-bit

Load/store ordered

These instructions are under Loads and Stores.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		0	0	1	0	0	0	1	L	0	Rs					o0	Rt2					Rn					Rt				

Decode fields			Instruction Details	Feature
size	L	o0		
00	0	0	STLLRB	FEAT_LOR
00	0	1	STLRB	-
00	1	0	LDLARB	FEAT_LOR
00	1	1	LDARB	-
01	0	0	STLLRH	FEAT_LOR
01	0	1	STLRH	-
01	1	0	LDLARH	FEAT_LOR
01	1	1	LDARH	-
10	0	0	STLLR — 32-bit	FEAT_LOR
10	0	1	STLR — 32-bit	-
10	1	0	LDLAR — 32-bit	FEAT_LOR
10	1	1	LDAR — 32-bit	-
11	0	0	STLLR — 64-bit	FEAT_LOR
11	0	1	STLR — 64-bit	-
11	1	0	LDLAR — 64-bit	FEAT_LOR
11	1	1	LDAR — 64-bit	-

Compare and swap

These instructions are under Loads and Stores.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		0	0	1	0	0	0	1	L	1	Rs					o0	Rt2					Rn					Rt				

Decode fields				Instruction Details	Feature
size	L	o0	Rt2		
			!= 11111	UNALLOCATED	-

Decode fields size	L	o0	Rt2	Instruction Details	Feature
00	0	0	11111	CASB, CASAB, CASALB, CASLB — CASB	FEAT_LSE
00	0	1	11111	CASB, CASAB, CASALB, CASLB — CASLB	FEAT_LSE
00	1	0	11111	CASB, CASAB, CASALB, CASLB — CASAB	FEAT_LSE
00	1	1	11111	CASB, CASAB, CASALB, CASLB — CASALB	FEAT_LSE
01	0	0	11111	CASH, CASAH, CASALH, CASLH — CASH	FEAT_LSE
01	0	1	11111	CASH, CASAH, CASALH, CASLH — CASLH	FEAT_LSE
01	1	0	11111	CASH, CASAH, CASALH, CASLH — CASAH	FEAT_LSE
01	1	1	11111	CASH, CASAH, CASALH, CASLH — CASALH	FEAT_LSE
10	0	0	11111	CAS, CASA, CASAL, CASL — 32-bit CAS	FEAT_LSE
10	0	1	11111	CAS, CASA, CASAL, CASL — 32-bit CASL	FEAT_LSE
10	1	0	11111	CAS, CASA, CASAL, CASL — 32-bit CASA	FEAT_LSE
10	1	1	11111	CAS, CASA, CASAL, CASL — 32-bit CASAL	FEAT_LSE
11	0	0	11111	CAS, CASA, CASAL, CASL — 64-bit CAS	FEAT_LSE
11	0	1	11111	CAS, CASA, CASAL, CASL — 64-bit CASL	FEAT_LSE
11	1	0	11111	CAS, CASA, CASAL, CASL — 64-bit CASA	FEAT_LSE
11	1	1	11111	CAS, CASA, CASAL, CASL — 64-bit CASAL	FEAT_LSE

LDAPR/STLR (unscaled immediate)

These instructions are under Loads and Stores.

31 30	29 28 27 26 25 24	23 22	21	20 19 18 17 16 15 14 13 12	11 10	9 8 7 6 5	4 3 2 1 0
size	0 1 1 0 0 1	opc	0	imm9	0 0	Rn	Rt

Decode fields size	opc	Instruction Details	Feature
00	00	STLURB	FEAT_LRCPC2
00	01	LDAPURB	FEAT_LRCPC2
00	10	LDAPURSB — 64-bit	FEAT_LRCPC2
00	11	LDAPURSB — 32-bit	FEAT_LRCPC2
01	00	STLURH	FEAT_LRCPC2
01	01	LDAPURH	FEAT_LRCPC2
01	10	LDAPURSH — 64-bit	FEAT_LRCPC2
01	11	LDAPURSH — 32-bit	FEAT_LRCPC2
10	00	STLUR — 32-bit	FEAT_LRCPC2
10	01	LDAPUR — 32-bit	FEAT_LRCPC2
10	10	LDAPURSW	FEAT_LRCPC2
10	11	UNALLOCATED	-
11	00	STLUR — 64-bit	FEAT_LRCPC2
11	01	LDAPUR — 64-bit	FEAT_LRCPC2
11	10	UNALLOCATED	-
11	11	UNALLOCATED	-

Load register (literal)

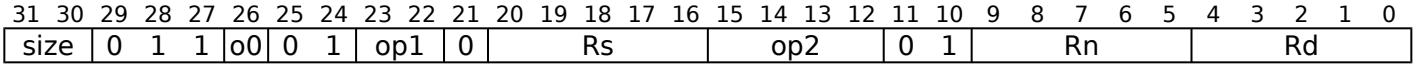
These instructions are under Loads and Stores.

31 30	29 28 27	26	25 24	23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5	4 3 2 1 0
opc	0 1 1	V	0 0	imm19	Rt

Decode fields opc	V	Instruction Details
00	0	LDR (literal) — 32-bit
00	1	LDR (literal, SIMD&FP) — 32-bit
01	0	LDR (literal) — 64-bit
01	1	LDR (literal, SIMD&FP) — 64-bit
10	0	LDRSW (literal)
10	1	LDR (literal, SIMD&FP) — 128-bit
11	0	PRFM (literal)
11	1	UNALLOCATED

Memory Copy and Memory Set

These instructions are under Loads and Stores.



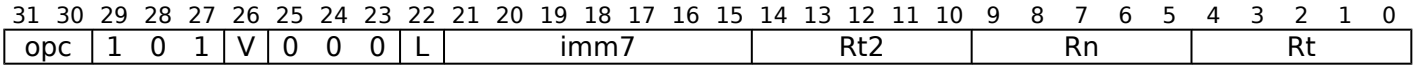
Decode fields o0	op1	op2	Instruction Details	Feature
0	00	0000	CPYFP, CPYFM, CPYFE — CPYFP	FEAT_MOPS
0	00	0001	CPYFPWT, CPYFMWT, CPYFEWT — CPYFPWT	FEAT_MOPS
0	00	0010	CPYFPRT, CPYFMRT, CPYFERT — CPYFPRT	FEAT_MOPS
0	00	0011	CPYFPT, CPYFMT, CPYFET — CPYFPT	FEAT_MOPS
0	00	0100	CPYFPWN, CPYFMWN, CPYFEWN — CPYFPWN	FEAT_MOPS
0	00	0101	CPYFPWTWN, CPYFMWTWN, CPYFEWTWN — CPYFPWTWN	FEAT_MOPS
0	00	0110	CPYFPRTWN, CPYFMRTWN, CPYFERTWN — CPYFPRTWN	FEAT_MOPS
0	00	0111	CPYFPTWN, CPYFMTWN, CPYFETWN — CPYFPTWN	FEAT_MOPS
0	00	1000	CPYFPRN, CPYFMRN, CPYFERN — CPYFPRN	FEAT_MOPS
0	00	1001	CPYFPWTRN, CPYFMWTRN, CPYFEWTRN — CPYFPWTRN	FEAT_MOPS
0	00	1010	CPYFPRTRN, CPYFMRTRN, CPYFERTRN — CPYFPRTRN	FEAT_MOPS
0	00	1011	CPYFPTRN, CPYFMTRN, CPYFETRN — CPYFPTRN	FEAT_MOPS
0	00	1100	CPYFPN, CPYFMN, CPYFEN — CPYFPN	FEAT_MOPS
0	00	1101	CPYFPWTN, CPYFMWTN, CPYFEWTN — CPYFPWTN	FEAT_MOPS
0	00	1110	CPYFPRTN, CPYFMRTN, CPYFERTN — CPYFPRTN	FEAT_MOPS
0	00	1111	CPYFPTN, CPYFMTN, CPYFETN — CPYFPTN	FEAT_MOPS
0	01	0000	CPYFP, CPYFM, CPYFE — CPYFM	FEAT_MOPS
0	01	0001	CPYFPWT, CPYFMWT, CPYFEWT — CPYFMWT	FEAT_MOPS
0	01	0010	CPYFPRT, CPYFMRT, CPYFERT — CPYFMRT	FEAT_MOPS
0	01	0011	CPYFPT, CPYFMT, CPYFET — CPYFMT	FEAT_MOPS
0	01	0100	CPYFPWN, CPYFMWN, CPYFEWN — CPYFMWN	FEAT_MOPS
0	01	0101	CPYFPWTWN, CPYFMWTWN, CPYFEWTWN — CPYFMWTWN	FEAT_MOPS
0	01	0110	CPYFPRTWN, CPYFMRTWN, CPYFERTWN — CPYFMRTWN	FEAT_MOPS
0	01	0111	CPYFPTWN, CPYFMTWN, CPYFETWN — CPYFMTWN	FEAT_MOPS
0	01	1000	CPYFPRN, CPYFMRN, CPYFERN — CPYFMRN	FEAT_MOPS
0	01	1001	CPYFPWTRN, CPYFMWTRN, CPYFEWTRN — CPYFMWTRN	FEAT_MOPS
0	01	1010	CPYFPRTRN, CPYFMRTRN, CPYFERTRN — CPYFMRTRN	FEAT_MOPS
0	01	1011	CPYFPTRN, CPYFMTRN, CPYFETRN — CPYFMTRN	FEAT_MOPS
0	01	1100	CPYFPN, CPYFMN, CPYFEN — CPYFMN	FEAT_MOPS
0	01	1101	CPYFPWTN, CPYFMWTN, CPYFEWTN — CPYFMWTN	FEAT_MOPS
0	01	1110	CPYFPRTN, CPYFMRTN, CPYFERTN — CPYFMRTN	FEAT_MOPS

Decode fields			Instruction Details	Feature
o0	op1	op2		
0	01	1111	CPYFPTN, CPYFMTN, CPYFETN — CPYFMTN	FEAT_MOPS
0	10	0000	CPYFP, CPYFM, CPYFE — CPYFE	FEAT_MOPS
0	10	0001	CPYFPWT, CPYFMWT, CPYFEWT — CPYFEWT	FEAT_MOPS
0	10	0010	CPYFPRT, CPYFMRT, CPYFERT — CPYFERT	FEAT_MOPS
0	10	0011	CPYFPT, CPYFMT, CPYFET — CPYFET	FEAT_MOPS
0	10	0100	CPYFPWN, CPYFMWN, CPYFEWN — CPYFEWN	FEAT_MOPS
0	10	0101	CPYFPWTWN, CPYFMWTWN, CPYFEWTWN — CPYFEWTWN	FEAT_MOPS
0	10	0110	CPYFPRTWN, CPYFMRTWN, CPYFERTWN — CPYFERTWN	FEAT_MOPS
0	10	0111	CPYFPTWN, CPYFMTWN, CPYFETWN — CPYFETWN	FEAT_MOPS
0	10	1000	CPYFPRN, CPYFMRN, CPYFERN — CPYFERN	FEAT_MOPS
0	10	1001	CPYFPWTRN, CPYFMWTRN, CPYFEWTRN — CPYFEWTRN	FEAT_MOPS
0	10	1010	CPYFPRTRN, CPYFMRTRN, CPYFERTRN — CPYFERTRN	FEAT_MOPS
0	10	1011	CPYFPTRN, CPYFMTRN, CPYFETRN — CPYFETRN	FEAT_MOPS
0	10	1100	CPYFPN, CPYFMN, CPYFEN — CPYFEN	FEAT_MOPS
0	10	1101	CPYFPWTN, CPYFMWTN, CPYFEWTN — CPYFEWTN	FEAT_MOPS
0	10	1110	CPYFPRTN, CPYFMRTN, CPYFERTN — CPYFERTN	FEAT_MOPS
0	10	1111	CPYFPTN, CPYFMTN, CPYFETN — CPYFETN	FEAT_MOPS
0	11	0000	SETP, SETM, SETE — SETP	FEAT_MOPS
0	11	0001	SETPT, SETMT, SETET — SETPT	FEAT_MOPS
0	11	0010	SETPN, SETMN, SETEN — SETPN	FEAT_MOPS
0	11	0011	SETPTN, SETMTN, SETETN — SETPTN	FEAT_MOPS
0	11	0100	SETP, SETM, SETE — SETM	FEAT_MOPS
0	11	0101	SETPT, SETMT, SETET — SETMT	FEAT_MOPS
0	11	0110	SETPN, SETMN, SETEN — SETMN	FEAT_MOPS
0	11	0111	SETPTN, SETMTN, SETETN — SETMTN	FEAT_MOPS
0	11	1000	SETP, SETM, SETE — SETE	FEAT_MOPS
0	11	1001	SETPT, SETMT, SETET — SETET	FEAT_MOPS
0	11	1010	SETPN, SETMN, SETEN — SETEN	FEAT_MOPS
0	11	1011	SETPTN, SETMTN, SETETN — SETETN	FEAT_MOPS
0	11	11xx	UNALLOCATED	-
1	00	0000	CPYP, CPYM, CPYE — CPYP	FEAT_MOPS
1	00	0001	CPYPWT, CPYMWT, CPYEWT — CPYPWT	FEAT_MOPS
1	00	0010	CPYPRT, CPYMRT, CPYERT — CPYPRT	FEAT_MOPS
1	00	0011	CPYPT, CPYMT, CPYET — CPYPT	FEAT_MOPS
1	00	0100	CPYPWN, CPYMWN, CPYEWN — CPYPWN	FEAT_MOPS
1	00	0101	CPYPWTWN, CPYMWTWN, CPYEWTWN — CPYPWTWN	FEAT_MOPS
1	00	0110	CPYPRTWN, CPYMRTWN, CPYERTWN — CPYPRTWN	FEAT_MOPS
1	00	0111	CPYPTWN, CPYMTWN, CPYETWN — CPYPTWN	FEAT_MOPS
1	00	1000	CPYPRN, CPYMRN, CPYERN — CPYPRN	FEAT_MOPS
1	00	1001	CPYPWTRN, CPYMWTRN, CPYEWTRN — CPYPWTRN	FEAT_MOPS
1	00	1010	CPYPRTRN, CPYMRTRN, CPYERTRN — CPYPRTRN	FEAT_MOPS
1	00	1011	CPYPTRN, CPYMTRN, CPYETRN — CPYPTRN	FEAT_MOPS
1	00	1100	CPYPN, CPYMN, CPYEN — CPYPN	FEAT_MOPS
1	00	1101	CPYPWTN, CPYMWTN, CPYEWTN — CPYPWTN	FEAT_MOPS
1	00	1110	CPYPRTN, CPYMRTN, CPYERTN — CPYPRTN	FEAT_MOPS
1	00	1111	CPYPTN, CPYMTN, CPYETN — CPYPTN	FEAT_MOPS
1	01	0000	CPYP, CPYM, CPYE — CPYM	FEAT_MOPS

Decode fields			Instruction Details	Feature
o0	op1	op2		
1	01	0001	CPYPWT, CPYMWT, CPYEWT — CPYMWT	FEAT_MOPS
1	01	0010	CPYPRT, CPYMRT, CPYERT — CPYMRT	FEAT_MOPS
1	01	0011	CPYPT, CPYMT, CPYET — CPYMT	FEAT_MOPS
1	01	0100	CPYPWN, CPYMWN, CPYEWN — CPYMWN	FEAT_MOPS
1	01	0101	CPYPWTWN, CPYMWTWN, CPYEWTWN — CPYMWTWN	FEAT_MOPS
1	01	0110	CPYPRTWN, CPYMRTWN, CPYERTWN — CPYMRTWN	FEAT_MOPS
1	01	0111	CPYPTWN, CPYMTWN, CPYETWN — CPYMTWN	FEAT_MOPS
1	01	1000	CPYPRN, CPYMRN, CPYERN — CPYMRN	FEAT_MOPS
1	01	1001	CPYPWTRN, CPYMWTRN, CPYEWTRN — CPYMWTRN	FEAT_MOPS
1	01	1010	CPYPRTRN, CPYMRTRN, CPYERTRN — CPYMRTRN	FEAT_MOPS
1	01	1011	CPYPTRN, CPYMTRN, CPYETRN — CPYMTRN	FEAT_MOPS
1	01	1100	CPYPN, CPYMN, CPYEN — CPYMN	FEAT_MOPS
1	01	1101	CPYPWTN, CPYMWTN, CPYEWTN — CPYMWTN	FEAT_MOPS
1	01	1110	CPYPRTN, CPYMRTN, CPYERTN — CPYMRTN	FEAT_MOPS
1	01	1111	CPYPTN, CPYMTN, CPYETN — CPYMTN	FEAT_MOPS
1	10	0000	CPYP, CPYM, CPYE — CPYE	FEAT_MOPS
1	10	0001	CPYPWT, CPYMWT, CPYEWT — CPYEWT	FEAT_MOPS
1	10	0010	CPYPRT, CPYMRT, CPYERT — CPYERT	FEAT_MOPS
1	10	0011	CPYPT, CPYMT, CPYET — CPYET	FEAT_MOPS
1	10	0100	CPYPWN, CPYMWN, CPYEWN — CPYEWN	FEAT_MOPS
1	10	0101	CPYPWTWN, CPYMWTWN, CPYEWTWN — CPYEWTWN	FEAT_MOPS
1	10	0110	CPYPRTWN, CPYMRTWN, CPYERTWN — CPYERTWN	FEAT_MOPS
1	10	0111	CPYPTWN, CPYMTWN, CPYETWN — CPYETWN	FEAT_MOPS
1	10	1000	CPYPRN, CPYMRN, CPYERN — CPYERN	FEAT_MOPS
1	10	1001	CPYPWTRN, CPYMWTRN, CPYEWTRN — CPYEWTRN	FEAT_MOPS
1	10	1010	CPYPRTRN, CPYMRTRN, CPYERTRN — CPYERTRN	FEAT_MOPS
1	10	1011	CPYPTRN, CPYMTRN, CPYETRN — CPYETRN	FEAT_MOPS
1	10	1100	CPYPN, CPYMN, CPYEN — CPYEN	FEAT_MOPS
1	10	1101	CPYPWTN, CPYMWTN, CPYEWTN — CPYEWTN	FEAT_MOPS
1	10	1110	CPYPRTN, CPYMRTN, CPYERTN — CPYERTN	FEAT_MOPS
1	10	1111	CPYPTN, CPYMTN, CPYETN — CPYETN	FEAT_MOPS
1	11	0000	SETGP, SETGM, SETGE — SETGP	FEAT_MOPS
1	11	0001	SETGPT, SETGMT, SETGET — SETGPT	FEAT_MOPS
1	11	0010	SETGPN, SETGMN, SETGEN — SETGPN	FEAT_MOPS
1	11	0011	SETGPTN, SETGMTN, SETGETN — SETGPTN	FEAT_MOPS
1	11	0100	SETGP, SETGM, SETGE — SETGM	FEAT_MOPS
1	11	0101	SETGPT, SETGMT, SETGET — SETGMT	FEAT_MOPS
1	11	0110	SETGPN, SETGMN, SETGEN — SETGMN	FEAT_MOPS
1	11	0111	SETGPTN, SETGMTN, SETGETN — SETGMTN	FEAT_MOPS
1	11	1000	SETGP, SETGM, SETGE — SETGE	FEAT_MOPS
1	11	1001	SETGPT, SETGMT, SETGET — SETGET	FEAT_MOPS
1	11	1010	SETGPN, SETGMN, SETGEN — SETGEN	FEAT_MOPS
1	11	1011	SETGPTN, SETGMTN, SETGETN — SETGETN	FEAT_MOPS
1	11	11xx	UNALLOCATED	-

Load/store no-allocate pair (offset)

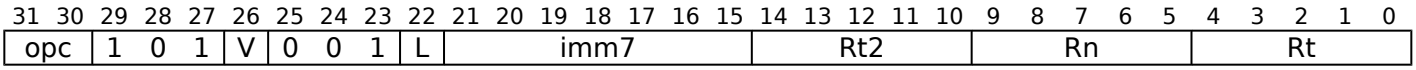
These instructions are under Loads and Stores.



Decode fields opc	V	L	Instruction Details
00	0	0	STNP — 32-bit
00	0	1	LDNP — 32-bit
00	1	0	STNP (SIMD&FP) — 32-bit
00	1	1	LDNP (SIMD&FP) — 32-bit
01	0		UNALLOCATED
01	1	0	STNP (SIMD&FP) — 64-bit
01	1	1	LDNP (SIMD&FP) — 64-bit
10	0	0	STNP — 64-bit
10	0	1	LDNP — 64-bit
10	1	0	STNP (SIMD&FP) — 128-bit
10	1	1	LDNP (SIMD&FP) — 128-bit
11			UNALLOCATED

Load/store register pair (post-indexed)

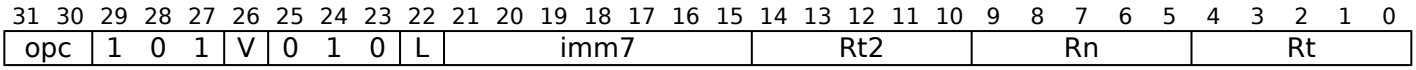
These instructions are under Loads and Stores.



Decode fields opc	V	L	Instruction Details	Feature
00	0	0	STP — 32-bit	-
00	0	1	LDP — 32-bit	-
00	1	0	STP (SIMD&FP) — 32-bit	-
00	1	1	LDP (SIMD&FP) — 32-bit	-
01	0	0	STGP	FEAT_MTE
01	0	1	LDPSW	-
01	1	0	STP (SIMD&FP) — 64-bit	-
01	1	1	LDP (SIMD&FP) — 64-bit	-
10	0	0	STP — 64-bit	-
10	0	1	LDP — 64-bit	-
10	1	0	STP (SIMD&FP) — 128-bit	-
10	1	1	LDP (SIMD&FP) — 128-bit	-
11			UNALLOCATED	-

Load/store register pair (offset)

These instructions are under Loads and Stores.



Decode fields opc	V	L	Instruction Details	Feature
00	0	0	STP — 32-bit	-
00	0	1	LDP — 32-bit	-
00	1	0	STP (SIMD&FP) — 32-bit	-
00	1	1	LDP (SIMD&FP) — 32-bit	-

Decode fields opc	V	L	Instruction Details	Feature
01	0	0	STGP	FEAT_MTE
01	0	1	LDPSW	-
01	1	0	STP (SIMD&FP) — 64-bit	-
01	1	1	LDP (SIMD&FP) — 64-bit	-
10	0	0	STP — 64-bit	-
10	0	1	LDP — 64-bit	-
10	1	0	STP (SIMD&FP) — 128-bit	-
10	1	1	LDP (SIMD&FP) — 128-bit	-
11			UNALLOCATED	-

Load/store register pair (pre-indexed)

These instructions are under Loads and Stores.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opc		1	0	1	V	0	1	1	L	imm7							Rt2					Rn					Rt				

Decode fields opc	V	L	Instruction Details	Feature
00	0	0	STP — 32-bit	-
00	0	1	LDP — 32-bit	-
00	1	0	STP (SIMD&FP) — 32-bit	-
00	1	1	LDP (SIMD&FP) — 32-bit	-
01	0	0	STGP	FEAT_MTE
01	0	1	LDPSW	-
01	1	0	STP (SIMD&FP) — 64-bit	-
01	1	1	LDP (SIMD&FP) — 64-bit	-
10	0	0	STP — 64-bit	-
10	0	1	LDP — 64-bit	-
10	1	0	STP (SIMD&FP) — 128-bit	-
10	1	1	LDP (SIMD&FP) — 128-bit	-
11			UNALLOCATED	-

Load/store register (unscaled immediate)

These instructions are under Loads and Stores.

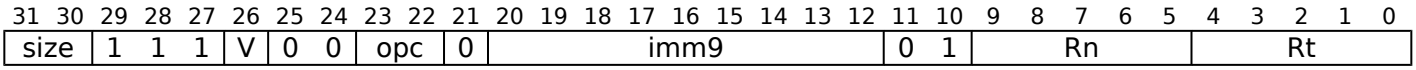
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	V	0	0	opc		0	imm9											0	0	Rn					Rt		

Decode fields size	V	opc	Instruction Details
x1	1	1x	UNALLOCATED
00	0	00	STURB
00	0	01	LDURB
00	0	10	LDURSB — 64-bit
00	0	11	LDURSB — 32-bit
00	1	00	STUR (SIMD&FP) — 8-bit
00	1	01	LDUR (SIMD&FP) — 8-bit
00	1	10	STUR (SIMD&FP) — 128-bit

Decode fields size	V	opc	Instruction Details
00	1	11	LDUR (SIMD&FP) — 128-bit
01	0	00	STURH
01	0	01	LDURH
01	0	10	LDURSH — 64-bit
01	0	11	LDURSH — 32-bit
01	1	00	STUR (SIMD&FP) — 16-bit
01	1	01	LDUR (SIMD&FP) — 16-bit
1x	0	11	UNALLOCATED
1x	1	1x	UNALLOCATED
10	0	00	STUR — 32-bit
10	0	01	LDUR — 32-bit
10	0	10	LDURSW
10	1	00	STUR (SIMD&FP) — 32-bit
10	1	01	LDUR (SIMD&FP) — 32-bit
11	0	00	STUR — 64-bit
11	0	01	LDUR — 64-bit
11	0	10	PRFUM
11	1	00	STUR (SIMD&FP) — 64-bit
11	1	01	LDUR (SIMD&FP) — 64-bit

Load/store register (immediate post-indexed)

These instructions are under Loads and Stores.

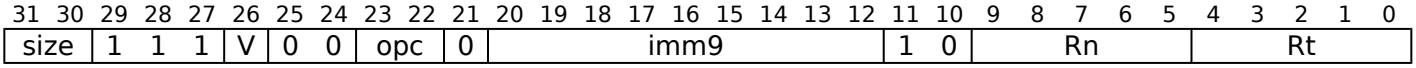


Decode fields size	V	opc	Instruction Details
x1	1	1x	UNALLOCATED
00	0	00	STRB (immediate)
00	0	01	LDRB (immediate)
00	0	10	LDRSB (immediate) — 64-bit
00	0	11	LDRSB (immediate) — 32-bit
00	1	00	STR (immediate, SIMD&FP) — 8-bit
00	1	01	LDR (immediate, SIMD&FP) — 8-bit
00	1	10	STR (immediate, SIMD&FP) — 128-bit
00	1	11	LDR (immediate, SIMD&FP) — 128-bit
01	0	00	STRH (immediate)
01	0	01	LDRH (immediate)
01	0	10	LDRSH (immediate) — 64-bit
01	0	11	LDRSH (immediate) — 32-bit
01	1	00	STR (immediate, SIMD&FP) — 16-bit
01	1	01	LDR (immediate, SIMD&FP) — 16-bit
1x	0	11	UNALLOCATED
1x	1	1x	UNALLOCATED
10	0	00	STR (immediate) — 32-bit
10	0	01	LDR (immediate) — 32-bit
10	0	10	LDRSW (immediate)

Decode fields size	V	opc	Instruction Details
10	1	00	STR (immediate, SIMD&FP) — 32-bit
10	1	01	LDR (immediate, SIMD&FP) — 32-bit
11	0	00	STR (immediate) — 64-bit
11	0	01	LDR (immediate) — 64-bit
11	0	10	UNALLOCATED
11	1	00	STR (immediate, SIMD&FP) — 64-bit
11	1	01	LDR (immediate, SIMD&FP) — 64-bit

Load/store register (unprivileged)

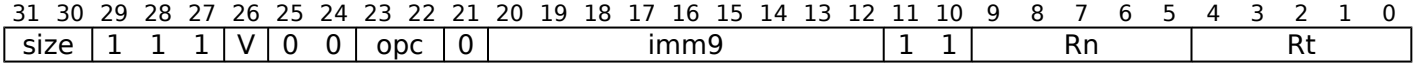
These instructions are under Loads and Stores.



Decode fields size	V	opc	Instruction Details
	1		UNALLOCATED
00	0	00	STTRB
00	0	01	LDTRB
00	0	10	LDTRSB — 64-bit
00	0	11	LDTRSB — 32-bit
01	0	00	STTRH
01	0	01	LDTRH
01	0	10	LDTRSH — 64-bit
01	0	11	LDTRSH — 32-bit
1x	0	11	UNALLOCATED
10	0	00	STTR — 32-bit
10	0	01	LDTR — 32-bit
10	0	10	LDTRSW
11	0	00	STTR — 64-bit
11	0	01	LDTR — 64-bit
11	0	10	UNALLOCATED

Load/store register (immediate pre-indexed)

These instructions are under Loads and Stores.

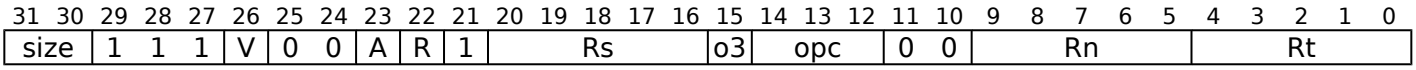


Decode fields size	V	opc	Instruction Details
x1	1	1x	UNALLOCATED
00	0	00	STRB (immediate)
00	0	01	LDRB (immediate)
00	0	10	LDRSB (immediate) — 64-bit
00	0	11	LDRSB (immediate) — 32-bit
00	1	00	STR (immediate, SIMD&FP) — 8-bit
00	1	01	LDR (immediate, SIMD&FP) — 8-bit

Decode fields			Instruction Details
size	V	opc	
00	1	10	STR (immediate, SIMD&FP) — 128-bit
00	1	11	LDR (immediate, SIMD&FP) — 128-bit
01	0	00	STRH (immediate)
01	0	01	LDRH (immediate)
01	0	10	LDRSH (immediate) — 64-bit
01	0	11	LDRSH (immediate) — 32-bit
01	1	00	STR (immediate, SIMD&FP) — 16-bit
01	1	01	LDR (immediate, SIMD&FP) — 16-bit
1x	0	11	UNALLOCATED
1x	1	1x	UNALLOCATED
10	0	00	STR (immediate) — 32-bit
10	0	01	LDR (immediate) — 32-bit
10	0	10	LDRSW (immediate)
10	1	00	STR (immediate, SIMD&FP) — 32-bit
10	1	01	LDR (immediate, SIMD&FP) — 32-bit
11	0	00	STR (immediate) — 64-bit
11	0	01	LDR (immediate) — 64-bit
11	0	10	UNALLOCATED
11	1	00	STR (immediate, SIMD&FP) — 64-bit
11	1	01	LDR (immediate, SIMD&FP) — 64-bit

Atomic memory operations

These instructions are under Loads and Stores.



Decode fields							Instruction Details	Feature
size	V	A	R	Rs	o3	opc		
	0				1	11x	UNALLOCATED	-
	0	0			1	100	UNALLOCATED	-
	0	0	1		1	001	UNALLOCATED	-
	0	0	1		1	010	UNALLOCATED	-
	0	0	1		1	011	UNALLOCATED	-
	0	0	1		1	101	UNALLOCATED	-
	0	1	0		1	001	UNALLOCATED	-
	0	1	0		1	010	UNALLOCATED	-
	0	1	0		1	011	UNALLOCATED	-
	0	1	0		1	101	UNALLOCATED	-
	0	1	1		1	001	UNALLOCATED	-
	0	1	1		1	010	UNALLOCATED	-
	0	1	1		1	011	UNALLOCATED	-
	0	1	1		1	100	UNALLOCATED	-
	0	1	1		1	101	UNALLOCATED	-
	1						UNALLOCATED	-
00	0	0	0		0	000	LDADDB, LDADDAB, LDADDALB, LDADDLB — LDADDB	FEAT_LSE

Decode fields							Instruction Details	Feature
size	V	A	R	Rs	o3	opc		
00	0	0	0		0	001	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB — LDCLRB	FEAT_LSE
00	0	0	0		0	010	LDEORB, LDEORAB, LDEORALB, LDEORLB — LDEORB	FEAT_LSE
00	0	0	0		0	011	LDSETB, LDSETAB, LDSETALB, LDSETLB — LDSETB	FEAT_LSE
00	0	0	0		0	100	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — LDSMAXB	FEAT_LSE
00	0	0	0		0	101	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — LDSMINB	FEAT_LSE
00	0	0	0		0	110	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — LDUMAXB	FEAT_LSE
00	0	0	0		0	111	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — LDUMINB	FEAT_LSE
00	0	0	0		1	000	SWPB, SWPAB, SWPALB, SWPLB — SWPB	FEAT_LSE
00	0	0	0		1	001	UNALLOCATED	-
00	0	0	0		1	010	UNALLOCATED	-
00	0	0	0		1	011	UNALLOCATED	-
00	0	0	0		1	101	UNALLOCATED	-
00	0	0	1		0	000	LDADDB, LDADDAB, LDADDALB, LDADDLB — LDADDLB	FEAT_LSE
00	0	0	1		0	001	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB — LDCLRLB	FEAT_LSE
00	0	0	1		0	010	LDEORB, LDEORAB, LDEORALB, LDEORLB — LDEORLB	FEAT_LSE
00	0	0	1		0	011	LDSETB, LDSETAB, LDSETALB, LDSETLB — LDSETLB	FEAT_LSE
00	0	0	1		0	100	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — LDSMAXLB	FEAT_LSE
00	0	0	1		0	101	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — LDSMINLB	FEAT_LSE
00	0	0	1		0	110	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — LDUMAXLB	FEAT_LSE
00	0	0	1		0	111	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — LDUMINLB	FEAT_LSE
00	0	0	1		1	000	SWPB, SWPAB, SWPALB, SWPLB — SWPLB	FEAT_LSE
00	0	1	0		0	000	LDADDB, LDADDAB, LDADDALB, LDADDLB — LDADDAB	FEAT_LSE
00	0	1	0		0	001	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB — LDCLRAB	FEAT_LSE
00	0	1	0		0	010	LDEORB, LDEORAB, LDEORALB, LDEORLB — LDEORAB	FEAT_LSE
00	0	1	0		0	011	LDSETB, LDSETAB, LDSETALB, LDSETLB — LDSETAB	FEAT_LSE
00	0	1	0		0	100	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — LDSMAXAB	FEAT_LSE
00	0	1	0		0	101	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — LDSMINAB	FEAT_LSE
00	0	1	0		0	110	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — LDUMAXAB	FEAT_LSE
00	0	1	0		0	111	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — LDUMINAB	FEAT_LSE
00	0	1	0		1	000	SWPB, SWPAB, SWPALB, SWPLB — SWPAB	FEAT_LSE

Decode fields							Instruction Details	Feature
size	V	A	R	Rs	o3	opc		
00	0	1	0		1	100	LDAPRB	FEAT_LRCPC
00	0	1	1		0	000	LDADDB, LDADDAB, LDADDALB, LDADDLB — LDADDALB	FEAT_LSE
00	0	1	1		0	001	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB — LDCLRALB	FEAT_LSE
00	0	1	1		0	010	LDEORB, LDEORAB, LDEORALB, LDEORLB — LDEORALB	FEAT_LSE
00	0	1	1		0	011	LDSETB, LDSETAB, LDSETALB, LDSETLB — LDSETALB	FEAT_LSE
00	0	1	1		0	100	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — LDSMAXALB	FEAT_LSE
00	0	1	1		0	101	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — LDSMINALB	FEAT_LSE
00	0	1	1		0	110	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — LDUMAXALB	FEAT_LSE
00	0	1	1		0	111	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — LDUMINALB	FEAT_LSE
00	0	1	1		1	000	SWPB, SWPAB, SWPALB, SWPLB — SWPALB	FEAT_LSE
01	0	0	0		0	000	LDADDH, LDADDAH, LDADDALH, LDADDLH — LDADDH	FEAT_LSE
01	0	0	0		0	001	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — LDCLRH	FEAT_LSE
01	0	0	0		0	010	LDEORH, LDEORAH, LDEORALH, LDEORLH — LDEORH	FEAT_LSE
01	0	0	0		0	011	LDSETH, LDSETAH, LDSETALH, LDSETLH — LDSETH	FEAT_LSE
01	0	0	0		0	100	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — LDSMAXH	FEAT_LSE
01	0	0	0		0	101	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — LDSMINH	FEAT_LSE
01	0	0	0		0	110	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — LDUMAXH	FEAT_LSE
01	0	0	0		0	111	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — LDUMINH	FEAT_LSE
01	0	0	0		1	000	SWPH, SWPAH, SWPALH, SWPLH — SWPH	FEAT_LSE
01	0	0	0		1	001	UNALLOCATED	-
01	0	0	0		1	010	UNALLOCATED	-
01	0	0	0		1	011	UNALLOCATED	-
01	0	0	0		1	101	UNALLOCATED	-
01	0	0	1		0	000	LDADDH, LDADDAH, LDADDALH, LDADDLH — LDADDLH	FEAT_LSE
01	0	0	1		0	001	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — LDCLRLH	FEAT_LSE
01	0	0	1		0	010	LDEORH, LDEORAH, LDEORALH, LDEORLH — LDEORLH	FEAT_LSE
01	0	0	1		0	011	LDSETH, LDSETAH, LDSETALH, LDSETLH — LDSETLH	FEAT_LSE
01	0	0	1		0	100	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — LDSMAXLH	FEAT_LSE
01	0	0	1		0	101	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — LDSMINLH	FEAT_LSE
01	0	0	1		0	110	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — LDUMAXLH	FEAT_LSE

Decode fields							Instruction Details	Feature
size	V	A	R	Rs	o3	opc		
01	0	0	1		0	111	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — LDUMINLH	FEAT_LSE
01	0	0	1		1	000	SWPH, SWPAH, SWPALH, SWPLH — SWPLH	FEAT_LSE
01	0	1	0		0	000	LDADDH, LDADDAH, LDADDALH, LDADDLH — LDADDAH	FEAT_LSE
01	0	1	0		0	001	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — LDCLRAH	FEAT_LSE
01	0	1	0		0	010	LDEORH, LDEORAH, LDEORALH, LDEORLH — LDEORAH	FEAT_LSE
01	0	1	0		0	011	LDSETH, LDSETAH, LDSETALH, LDSETLH — LDSETAH	FEAT_LSE
01	0	1	0		0	100	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — LDSMAXAH	FEAT_LSE
01	0	1	0		0	101	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — LDSMINAH	FEAT_LSE
01	0	1	0		0	110	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — LDUMAXAH	FEAT_LSE
01	0	1	0		0	111	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — LDUMINAH	FEAT_LSE
01	0	1	0		1	000	SWPH, SWPAH, SWPALH, SWPLH — SWPAH	FEAT_LSE
01	0	1	0		1	100	LDAPRH	FEAT_LRCPC
01	0	1	1		0	000	LDADDH, LDADDAH, LDADDALH, LDADDLH — LDADDALH	FEAT_LSE
01	0	1	1		0	001	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — LDCLRALH	FEAT_LSE
01	0	1	1		0	010	LDEORH, LDEORAH, LDEORALH, LDEORLH — LDEORALH	FEAT_LSE
01	0	1	1		0	011	LDSETH, LDSETAH, LDSETALH, LDSETLH — LDSETALH	FEAT_LSE
01	0	1	1		0	100	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — LDSMAXALH	FEAT_LSE
01	0	1	1		0	101	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — LDSMINALH	FEAT_LSE
01	0	1	1		0	110	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — LDUMAXALH	FEAT_LSE
01	0	1	1		0	111	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — LDUMINALH	FEAT_LSE
01	0	1	1		1	000	SWPH, SWPAH, SWPALH, SWPLH — SWPALH	FEAT_LSE
10	0	0	0		0	000	LDADD, LDADDA, LDADDAL, LDADDL — 32-bit LDADD	FEAT_LSE
10	0	0	0		0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit LDCLR	FEAT_LSE
10	0	0	0		0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit LDEOR	FEAT_LSE
10	0	0	0		0	011	LDSET, LDSETA, LDSETAL, LDSETL — 32-bit LDSET	FEAT_LSE
10	0	0	0		0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit LDSMAX	FEAT_LSE
10	0	0	0		0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit LDSMIN	FEAT_LSE
10	0	0	0		0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit LDUMAX	FEAT_LSE
10	0	0	0		0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit LDUMIN	FEAT_LSE

Decode fields							Instruction Details	Feature
size	V	A	R	Rs	o3	opc		
10	0	0	0		1	000	SWP, SWPA, SWPAL, SWPL — 32-bit SWP	FEAT_LSE
10	0	0	0		1	001	UNALLOCATED	-
10	0	0	0		1	010	UNALLOCATED	-
10	0	0	0		1	011	UNALLOCATED	-
10	0	0	0		1	101	UNALLOCATED	-
10	0	0	1		0	000	LDADD, LDADDA, LDADDAL, LDADDL — 32-bit LDADDL	FEAT_LSE
10	0	0	1		0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit LDCLRL	FEAT_LSE
10	0	0	1		0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit LDEORL	FEAT_LSE
10	0	0	1		0	011	LDSET, LDSETA, LDSETAL, LDSETL — 32-bit LDSETL	FEAT_LSE
10	0	0	1		0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit LDSMAXL	FEAT_LSE
10	0	0	1		0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit LDSMINL	FEAT_LSE
10	0	0	1		0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit LDUMAXL	FEAT_LSE
10	0	0	1		0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit LDUMINL	FEAT_LSE
10	0	0	1		1	000	SWP, SWPA, SWPAL, SWPL — 32-bit SWPL	FEAT_LSE
10	0	1	0		0	000	LDADD, LDADDA, LDADDAL, LDADDL — 32-bit LDADDA	FEAT_LSE
10	0	1	0		0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit LDCLRA	FEAT_LSE
10	0	1	0		0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit LDEORA	FEAT_LSE
10	0	1	0		0	011	LDSET, LDSETA, LDSETAL, LDSETL — 32-bit LDSETA	FEAT_LSE
10	0	1	0		0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit LDSMAXA	FEAT_LSE
10	0	1	0		0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit LDSMINA	FEAT_LSE
10	0	1	0		0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit LDUMAXA	FEAT_LSE
10	0	1	0		0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit LDUMINA	FEAT_LSE
10	0	1	0		1	000	SWP, SWPA, SWPAL, SWPL — 32-bit SWPA	FEAT_LSE
10	0	1	0		1	100	LDAPR — 32-bit	FEAT_LRCPC
10	0	1	1		0	000	LDADD, LDADDA, LDADDAL, LDADDL — 32-bit LDADDAL	FEAT_LSE
10	0	1	1		0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit LDCLRAL	FEAT_LSE
10	0	1	1		0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit LDEORAL	FEAT_LSE
10	0	1	1		0	011	LDSET, LDSETA, LDSETAL, LDSETL — 32-bit LDSETAL	FEAT_LSE
10	0	1	1		0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit LDSMAXAL	FEAT_LSE
10	0	1	1		0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit LDSMINAL	FEAT_LSE

Decode fields							Instruction Details	Feature
size	V	A	R	Rs	o3	opc		
10	0	1	1		0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit LDUMAXAL	FEAT_LSE
10	0	1	1		0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit LDUMINAL	FEAT_LSE
10	0	1	1		1	000	SWP, SWPA, SWPAL, SWPL — 32-bit SWPAL	FEAT_LSE
11	0	0	0		0	000	LDADD, LDADDA, LDADDAL, LDADDL — 64-bit LDADD	FEAT_LSE
11	0	0	0		0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit LDCLR	FEAT_LSE
11	0	0	0		0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit LDEOR	FEAT_LSE
11	0	0	0		0	011	LDSET, LDSETA, LDSETAL, LDSETL — 64-bit LDSET	FEAT_LSE
11	0	0	0		0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit LDSMAX	FEAT_LSE
11	0	0	0		0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit LDSMIN	FEAT_LSE
11	0	0	0		0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit LDUMAX	FEAT_LSE
11	0	0	0		0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit LDUMIN	FEAT_LSE
11	0	0	0		1	000	SWP, SWPA, SWPAL, SWPL — 64-bit SWP	FEAT_LSE
11	0	0	0		1	010	ST64BV0	FEAT_LS64_ACCDATA
11	0	0	0		1	011	ST64BV	FEAT_LS64_V
11	0	0	0	11111	1	001	ST64B	FEAT_LS64
11	0	0	0	11111	1	101	LD64B	FEAT_LS64
11	0	0	1		0	000	LDADD, LDADDA, LDADDAL, LDADDL — 64-bit LDADDL	FEAT_LSE
11	0	0	1		0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit LDCLRL	FEAT_LSE
11	0	0	1		0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit LDEORL	FEAT_LSE
11	0	0	1		0	011	LDSET, LDSETA, LDSETAL, LDSETL — 64-bit LDSETL	FEAT_LSE
11	0	0	1		0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit LDSMAXL	FEAT_LSE
11	0	0	1		0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit LDSMINL	FEAT_LSE
11	0	0	1		0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit LDUMAXL	FEAT_LSE
11	0	0	1		0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit LDUMINL	FEAT_LSE
11	0	0	1		1	000	SWP, SWPA, SWPAL, SWPL — 64-bit SWPL	FEAT_LSE
11	0	1	0		0	000	LDADD, LDADDA, LDADDAL, LDADDL — 64-bit LDADDA	FEAT_LSE
11	0	1	0		0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit LDCLRA	FEAT_LSE
11	0	1	0		0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit LDEORA	FEAT_LSE
11	0	1	0		0	011	LDSET, LDSETA, LDSETAL, LDSETL — 64-bit LDSETA	FEAT_LSE
11	0	1	0		0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit LDSMAXA	FEAT_LSE

Decode fields							Instruction Details	Feature
size	V	A	R	Rs	o3	opc		
11	0	1	0		0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit LDSMINA	FEAT_LSE
11	0	1	0		0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit LDUMAXA	FEAT_LSE
11	0	1	0		0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit LDUMINA	FEAT_LSE
11	0	1	0		1	000	SWP, SWPA, SWPAL, SWPL — 64-bit SWPA	FEAT_LSE
11	0	1	0		1	100	LDAPR — 64-bit	FEAT_LRCPC
11	0	1	1		0	000	LDADD, LDADDA, LDADDAL, LDADDL — 64-bit LDADDAL	FEAT_LSE
11	0	1	1		0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit LDCLRAL	FEAT_LSE
11	0	1	1		0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit LDEORAL	FEAT_LSE
11	0	1	1		0	011	LDSET, LDSETA, LDSETAL, LDSETL — 64-bit LDSETAL	FEAT_LSE
11	0	1	1		0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit LDSMAXAL	FEAT_LSE
11	0	1	1		0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit LDSMINAL	FEAT_LSE
11	0	1	1		0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit LDUMAXAL	FEAT_LSE
11	0	1	1		0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit LDUMINAL	FEAT_LSE
11	0	1	1		1	000	SWP, SWPA, SWPAL, SWPL — 64-bit SWPAL	FEAT_LSE

Load/store register (register offset)

These instructions are under Loads and Stores.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	V	0	0	opc		1	Rm					option			S	1	0	Rn					Rt				

Decode fields				Instruction Details
size	V	opc	option	
x1	1	1x		UNALLOCATED
00	0	00	!= 011	STRB (register) — extended register
00	0	00	011	STRB (register) — shifted register
00	0	01	!= 011	LDRB (register) — extended register
00	0	01	011	LDRB (register) — shifted register
00	0	10	!= 011	LDRSB (register) — 64-bit with extended register offset
00	0	10	011	LDRSB (register) — 64-bit with shifted register offset
00	0	11	!= 011	LDRSB (register) — 32-bit with extended register offset
00	0	11	011	LDRSB (register) — 32-bit with shifted register offset
00	1	00	!= 011	STR (register, SIMD&FP)
00	1	00	011	STR (register, SIMD&FP)
00	1	01	!= 011	LDR (register, SIMD&FP)
00	1	01	011	LDR (register, SIMD&FP)
00	1	10		STR (register, SIMD&FP)
00	1	11		LDR (register, SIMD&FP)
01	0	00		STRH (register)

Decode fields				Instruction Details
size	V	opc	option	
01	0	01		LDRH (register)
01	0	10		LDRSH (register) — 64-bit
01	0	11		LDRSH (register) — 32-bit
01	1	00		STR (register, SIMD&FP)
01	1	01		LDR (register, SIMD&FP)
1x	0	11		UNALLOCATED
1x	1	1x		UNALLOCATED
10	0	00		STR (register) — 32-bit
10	0	01		LDR (register) — 32-bit
10	0	10		LDRSW (register)
10	1	00		STR (register, SIMD&FP)
10	1	01		LDR (register, SIMD&FP)
11	0	00		STR (register) — 64-bit
11	0	01		LDR (register) — 64-bit
11	0	10		PRFM (register)
11	1	00		STR (register, SIMD&FP)
11	1	01		LDR (register, SIMD&FP)

Load/store register (pac)

These instructions are under Loads and Stores.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	V	0	0	M	S	1	imm9									W	1	Rn					Rt				

Decode fields				Instruction Details	Feature
size	V	M	W		
!= 11				UNALLOCATED	-
11	0	0	0	LDRAA, LDRAB — key A, offset	FEAT_PAuth
11	0	0	1	LDRAA, LDRAB — key A, pre-indexed	FEAT_PAuth
11	0	1	0	LDRAA, LDRAB — key B, offset	FEAT_PAuth
11	0	1	1	LDRAA, LDRAB — key B, pre-indexed	FEAT_PAuth
11	1			UNALLOCATED	-

Load/store register (unsigned immediate)

These instructions are under Loads and Stores.

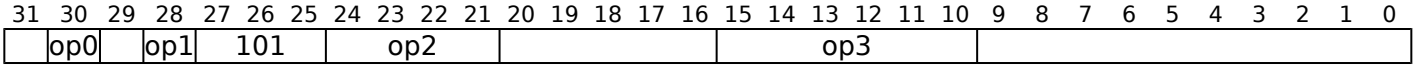
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
size		1	1	1	V	0	1	opc		imm12												Rn					Rt				

Decode fields			Instruction Details
size	V	opc	
x1	1	1x	UNALLOCATED
00	0	00	STRB (immediate)
00	0	01	LDRB (immediate)
00	0	10	LDRSB (immediate) — 64-bit
00	0	11	LDRSB (immediate) — 32-bit
00	1	00	STR (immediate, SIMD&FP) — 8-bit
00	1	01	LDR (immediate, SIMD&FP) — 8-bit

Decode fields size	V	opc	Instruction Details
00	1	10	STR (immediate, SIMD&FP) — 128-bit
00	1	11	LDR (immediate, SIMD&FP) — 128-bit
01	0	00	STRH (immediate)
01	0	01	LDRH (immediate)
01	0	10	LDRSH (immediate) — 64-bit
01	0	11	LDRSH (immediate) — 32-bit
01	1	00	STR (immediate, SIMD&FP) — 16-bit
01	1	01	LDR (immediate, SIMD&FP) — 16-bit
1x	0	11	UNALLOCATED
1x	1	1x	UNALLOCATED
10	0	00	STR (immediate) — 32-bit
10	0	01	LDR (immediate) — 32-bit
10	0	10	LDRSW (immediate)
10	1	00	STR (immediate, SIMD&FP) — 32-bit
10	1	01	LDR (immediate, SIMD&FP) — 32-bit
11	0	00	STR (immediate) — 64-bit
11	0	01	LDR (immediate) — 64-bit
11	0	10	PRFM (immediate)
11	1	00	STR (immediate, SIMD&FP) — 64-bit
11	1	01	LDR (immediate, SIMD&FP) — 64-bit

Data Processing -- Register

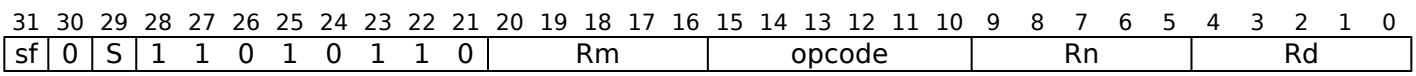
These instructions are under the top-level.



Decode fields op0	op1	op2	op3	Instruction details
0	1	0110		Data-processing (2 source)
1	1	0110		Data-processing (1 source)
	0	0xxx		Logical (shifted register)
	0	1xx0		Add/subtract (shifted register)
	0	1xx1		Add/subtract (extended register)
	1	0000	000000	Add/subtract (with carry)
	1	0000	x00001	Rotate right into flags
	1	0000	xx0010	Evaluate into flags
	1	0010	xxxx0x	Conditional compare (register)
	1	0010	xxxx1x	Conditional compare (immediate)
	1	0100		Conditional select
	1	1xxx		Data-processing (3 source)

Data-processing (2 source)

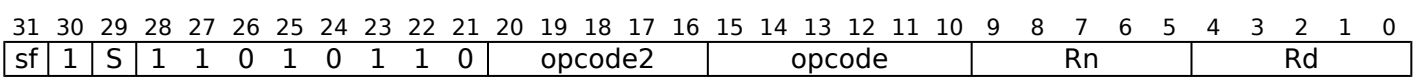
These instructions are under Data Processing -- Register.



Decode fields sf	S	opcode	Instruction Details	Feature
		000001	UNALLOCATED	-
		011xxx	UNALLOCATED	-
		1xxxxx	UNALLOCATED	-
	0	00011x	UNALLOCATED	-
	0	001101	UNALLOCATED	-
	0	00111x	UNALLOCATED	-
	1	00001x	UNALLOCATED	-
	1	0001xx	UNALLOCATED	-
	1	001xxx	UNALLOCATED	-
	1	01xxxx	UNALLOCATED	-
0		000000	UNALLOCATED	-
0	0	000010	UDIV — 32-bit	-
0	0	000011	SDIV — 32-bit	-
0	0	00010x	UNALLOCATED	-
0	0	001000	LSLV — 32-bit	-
0	0	001001	LSRV — 32-bit	-
0	0	001010	ASRV — 32-bit	-
0	0	001011	RORV — 32-bit	-
0	0	001100	UNALLOCATED	-
0	0	010x11	UNALLOCATED	-
0	0	010000	CRC32B, CRC32H, CRC32W, CRC32X — CRC32B	FEAT_CRC32
0	0	010001	CRC32B, CRC32H, CRC32W, CRC32X — CRC32H	FEAT_CRC32
0	0	010010	CRC32B, CRC32H, CRC32W, CRC32X — CRC32W	FEAT_CRC32
0	0	010100	CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CB	FEAT_CRC32
0	0	010101	CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CH	FEAT_CRC32
0	0	010110	CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CW	FEAT_CRC32
1	0	000000	SUBP	FEAT_MTE
1	0	000010	UDIV — 64-bit	-
1	0	000011	SDIV — 64-bit	-
1	0	000100	IRG	FEAT_MTE
1	0	000101	GMI	FEAT_MTE
1	0	001000	LSLV — 64-bit	-
1	0	001001	LSRV — 64-bit	-
1	0	001010	ASRV — 64-bit	-
1	0	001011	RORV — 64-bit	-
1	0	001100	PACGA	FEAT_PAuth
1	0	010xx0	UNALLOCATED	-
1	0	010x0x	UNALLOCATED	-
1	0	010011	CRC32B, CRC32H, CRC32W, CRC32X — CRC32X	FEAT_CRC32
1	0	010111	CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CX	FEAT_CRC32
1	1	000000	SUBPS	FEAT_MTE

Data-processing (1 source)

These instructions are under Data Processing -- Register.



Decode fields					Instruction Details	Feature
sf	S	opcode2	opcode	Rn		
			1xxxxx		UNALLOCATED	-
		xxx1x			UNALLOCATED	-
		xx1xx			UNALLOCATED	-
		x1xxx			UNALLOCATED	-
		1xxxx			UNALLOCATED	-
	0	00000	00011x		UNALLOCATED	-
	0	00000	001xxx		UNALLOCATED	-
	0	00000	01xxxx		UNALLOCATED	-
	1				UNALLOCATED	-
0		00001			UNALLOCATED	-
0	0	00000	000000		RBIT — 32-bit	-
0	0	00000	000001		REV16 — 32-bit	-
0	0	00000	000010		REV — 32-bit	-
0	0	00000	000011		UNALLOCATED	-
0	0	00000	000100		CLZ — 32-bit	-
0	0	00000	000101		CLS — 32-bit	-
1	0	00000	000000		RBIT — 64-bit	-
1	0	00000	000001		REV16 — 64-bit	-
1	0	00000	000010		REV32	-
1	0	00000	000011		REV — 64-bit	-
1	0	00000	000100		CLZ — 64-bit	-
1	0	00000	000101		CLS — 64-bit	-
1	0	00001	000000		PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIA	FEAT_PAuth
1	0	00001	000001		PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIB	FEAT_PAuth
1	0	00001	000010		PACDA, PACDZA — PACDA	FEAT_PAuth
1	0	00001	000011		PACDB, PACDZB — PACDB	FEAT_PAuth
1	0	00001	000100		AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIA	FEAT_PAuth
1	0	00001	000101		AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIB	FEAT_PAuth
1	0	00001	000110		AUTDA, AUTDZA — AUTDA	FEAT_PAuth
1	0	00001	000111		AUTDB, AUTDZB — AUTDB	FEAT_PAuth
1	0	00001	001000	11111	PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA — PACIZA	FEAT_PAuth
1	0	00001	001001	11111	PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB — PACIZB	FEAT_PAuth
1	0	00001	001010	11111	PACDA, PACDZA — PACDZA	FEAT_PAuth
1	0	00001	001011	11111	PACDB, PACDZB — PACDZB	FEAT_PAuth
1	0	00001	001100	11111	AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA — AUTIZA	FEAT_PAuth
1	0	00001	001101	11111	AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB — AUTIZB	FEAT_PAuth
1	0	00001	001110	11111	AUTDA, AUTDZA — AUTDZA	FEAT_PAuth
1	0	00001	001111	11111	AUTDB, AUTDZB — AUTDZB	FEAT_PAuth
1	0	00001	010000	11111	XPACD, XPACI, XPACLRI — XPACI	FEAT_PAuth
1	0	00001	010001	11111	XPACD, XPACI, XPACLRI — XPACD	FEAT_PAuth
1	0	00001	01001x		UNALLOCATED	-
1	0	00001	0101xx		UNALLOCATED	-

Decode fields					Instruction Details	Feature
sf	S	opcode2	opcode	Rn		
1	0	00001	011xxx		UNALLOCATED	-

Logical (shifted register)

These instructions are under Data Processing -- Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	opc		0	1	0	1	0	shift		N	Rm					imm6						Rn					Rd				

Decode fields				Instruction Details
sf	opc	N	imm6	
0			1xxxxx	UNALLOCATED
0	00	0		AND (shifted register) — 32-bit
0	00	1		BIC (shifted register) — 32-bit
0	01	0		ORR (shifted register) — 32-bit
0	01	1		ORN (shifted register) — 32-bit
0	10	0		EOR (shifted register) — 32-bit
0	10	1		EON (shifted register) — 32-bit
0	11	0		ANDS (shifted register) — 32-bit
0	11	1		BICS (shifted register) — 32-bit
1	00	0		AND (shifted register) — 64-bit
1	00	1		BIC (shifted register) — 64-bit
1	01	0		ORR (shifted register) — 64-bit
1	01	1		ORN (shifted register) — 64-bit
1	10	0		EOR (shifted register) — 64-bit
1	10	1		EON (shifted register) — 64-bit
1	11	0		ANDS (shifted register) — 64-bit
1	11	1		BICS (shifted register) — 64-bit

Add/subtract (shifted register)

These instructions are under Data Processing -- Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op	S	0	1	0	1	1	shift		0	Rm					imm6						Rn					Rd				

Decode fields					Instruction Details
sf	op	S	shift	imm6	
			11		UNALLOCATED
0				1xxxxx	UNALLOCATED
0	0	0			ADD (shifted register) — 32-bit
0	0	1			ADDS (shifted register) — 32-bit
0	1	0			SUB (shifted register) — 32-bit
0	1	1			SUBS (shifted register) — 32-bit
1	0	0			ADD (shifted register) — 64-bit
1	0	1			ADDS (shifted register) — 64-bit
1	1	0			SUB (shifted register) — 64-bit
1	1	1			SUBS (shifted register) — 64-bit

Add/subtract (extended register)

These instructions are under Data Processing -- Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op	S	0	1	0	1	1	opt		1	Rm					option			imm3			Rn					Rd				

Decode fields sf	op	S	opt	imm3	Instruction Details
				1x1	UNALLOCATED
				11x	UNALLOCATED
			x1		UNALLOCATED
			1x		UNALLOCATED
0	0	0	00		ADD (extended register) — 32-bit
0	0	1	00		ADDS (extended register) — 32-bit
0	1	0	00		SUB (extended register) — 32-bit
0	1	1	00		SUBS (extended register) — 32-bit
1	0	0	00		ADD (extended register) — 64-bit
1	0	1	00		ADDS (extended register) — 64-bit
1	1	0	00		SUB (extended register) — 64-bit
1	1	1	00		SUBS (extended register) — 64-bit

Add/subtract (with carry)

These instructions are under Data Processing -- Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op	S	1	1	0	1	0	0	0	0	Rm					0	0	0	0	0	0	Rn					Rd				

Decode fields sf	op	S	Instruction Details
0	0	0	ADC — 32-bit
0	0	1	ADCS — 32-bit
0	1	0	SBC — 32-bit
0	1	1	SBCS — 32-bit
1	0	0	ADC — 64-bit
1	0	1	ADCS — 64-bit
1	1	0	SBC — 64-bit
1	1	1	SBCS — 64-bit

Rotate right into flags

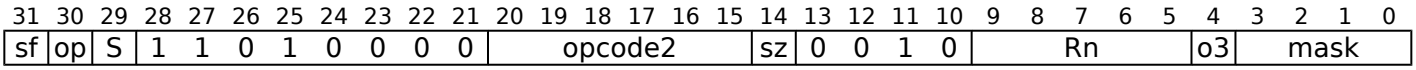
These instructions are under Data Processing -- Register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	op	S	1	1	0	1	0	0	0	0	imm6						0	0	0	0	1	Rn					o2	mask				

Decode fields sf	op	S	o2	Instruction Details	Feature
0				UNALLOCATED	-
1	0	0		UNALLOCATED	-
1	0	1	0	RMIF	FEAT_FlagM
1	0	1	1	UNALLOCATED	-
1	1			UNALLOCATED	-

Evaluate into flags

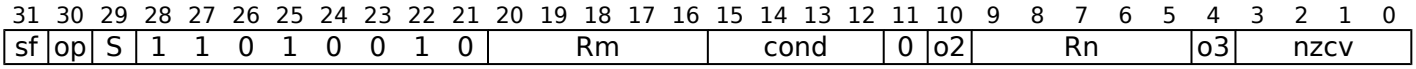
These instructions are under Data Processing -- Register.



Decode fields							Instruction Details	Feature
sf	op	S	opcode2	sz	o3	mask		
0	0	0					UNALLOCATED	-
0	0	1	!= 000000				UNALLOCATED	-
0	0	1	000000		0	!= 1101	UNALLOCATED	-
0	0	1	000000		1		UNALLOCATED	-
0	0	1	000000	0	0	1101	SETF8, SETF16 — SETF8	FEAT_FlagM
0	0	1	000000	1	0	1101	SETF8, SETF16 — SETF16	FEAT_FlagM
0	1						UNALLOCATED	-
1							UNALLOCATED	-

Conditional compare (register)

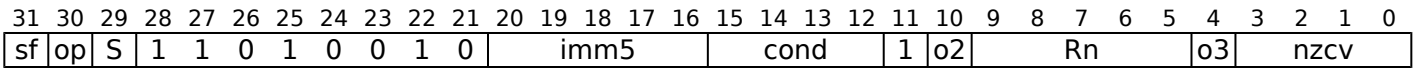
These instructions are under Data Processing -- Register.



Decode fields					Instruction Details
sf	op	S	o2	o3	
				1	UNALLOCATED
			1		UNALLOCATED
		0			UNALLOCATED
0	0	1	0	0	CCMN (register) — 32-bit
0	1	1	0	0	CCMP (register) — 32-bit
1	0	1	0	0	CCMN (register) — 64-bit
1	1	1	0	0	CCMP (register) — 64-bit

Conditional compare (immediate)

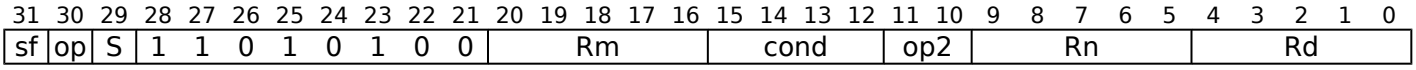
These instructions are under Data Processing -- Register.



Decode fields					Instruction Details
sf	op	S	o2	o3	
				1	UNALLOCATED
			1		UNALLOCATED
		0			UNALLOCATED
0	0	1	0	0	CCMN (immediate) — 32-bit
0	1	1	0	0	CCMP (immediate) — 32-bit
1	0	1	0	0	CCMN (immediate) — 64-bit
1	1	1	0	0	CCMP (immediate) — 64-bit

Conditional select

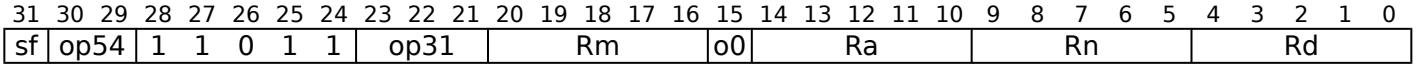
These instructions are under Data Processing -- Register.



sf	Decode fields op	S	op2	Instruction Details
			1x	UNALLOCATED
		1		UNALLOCATED
0	0	0	00	CSEL — 32-bit
0	0	0	01	CSINC — 32-bit
0	1	0	00	CSINV — 32-bit
0	1	0	01	CSNEG — 32-bit
1	0	0	00	CSEL — 64-bit
1	0	0	01	CSINC — 64-bit
1	1	0	00	CSINV — 64-bit
1	1	0	01	CSNEG — 64-bit

Data-processing (3 source)

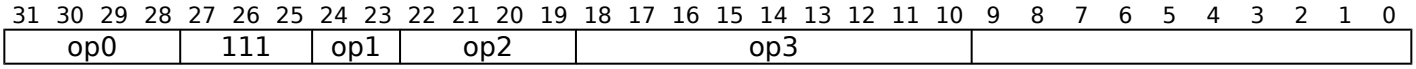
These instructions are under Data Processing -- Register.



sf	Decode fields op54	op31	o0	Instruction Details
	00	010	1	UNALLOCATED
	00	011		UNALLOCATED
	00	100		UNALLOCATED
	00	110	1	UNALLOCATED
	00	111		UNALLOCATED
	01			UNALLOCATED
	1x			UNALLOCATED
0	00	000	0	MADD — 32-bit
0	00	000	1	MSUB — 32-bit
0	00	001	0	UNALLOCATED
0	00	001	1	UNALLOCATED
0	00	010	0	UNALLOCATED
0	00	101	0	UNALLOCATED
0	00	101	1	UNALLOCATED
0	00	110	0	UNALLOCATED
1	00	000	0	MADD — 64-bit
1	00	000	1	MSUB — 64-bit
1	00	001	0	SMADDL
1	00	001	1	SMSUBL
1	00	010	0	SMULH
1	00	101	0	UMADDL
1	00	101	1	UMSUBL
1	00	110	0	UMULH

Data Processing -- Scalar Floating-Point and Advanced SIMD

These instructions are under the top-level.



Decode fields				Instruction details	Architecture version
op0	op1	op2	op3		
0000	0x	x101	00xxxxx10	UNALLOCATED	-
0010	0x	x101	00xxxxx10	UNALLOCATED	-
0100	0x	x101	00xxxxx10	Cryptographic AES	-
0101	0x	x0xx	xxx0xxx00	Cryptographic three-register SHA	-
0101	0x	x0xx	xxx0xxx10	UNALLOCATED	-
0101	0x	x101	00xxxxx10	Cryptographic two-register SHA	-
0110	0x	x101	00xxxxx10	UNALLOCATED	-
0111	0x	x0xx	xxx0xxxx0	UNALLOCATED	-
0111	0x	x101	00xxxxx10	UNALLOCATED	-
01x1	00	00xx	xxx0xxxx1	Advanced SIMD scalar copy	-
01x1	01	00xx	xxx0xxxx1	UNALLOCATED	-
01x1	0x	0111	00xxxxx10	UNALLOCATED	-
01x1	0x	10xx	xxx00xxx1	Advanced SIMD scalar three same FP16	-
01x1	0x	10xx	xxx01xxx1	UNALLOCATED	-
01x1	0x	1111	00xxxxx10	Advanced SIMD scalar two-register miscellaneous FP16	-
01x1	0x	x0xx	xxx1xxxx0	UNALLOCATED	-
01x1	0x	x0xx	xxx1xxxx1	Advanced SIMD scalar three same extra	-
01x1	0x	x100	00xxxxx10	Advanced SIMD scalar two-register miscellaneous	-
01x1	0x	x110	00xxxxx10	Advanced SIMD scalar pairwise	-
01x1	0x	x1xx	1xxxxxx10	UNALLOCATED	-
01x1	0x	x1xx	x1xxxxx10	UNALLOCATED	-
01x1	0x	x1xx	xxxxxxx00	Advanced SIMD scalar three different	-
01x1	0x	x1xx	xxxxxxxx1	Advanced SIMD scalar three same	-
01x1	10		xxxxxxxx1	Advanced SIMD scalar shift by immediate	-
01x1	11		xxxxxxxx1	UNALLOCATED	-
01x1	1x		xxxxxxxx0	Advanced SIMD scalar x indexed element	-
0x00	0x	x0xx	xxx0xxx00	Advanced SIMD table lookup	-
0x00	0x	x0xx	xxx0xxx10	Advanced SIMD permute	-
0x10	0x	x0xx	xxx0xxxx0	Advanced SIMD extract	-
0xx0	00	00xx	xxx0xxxx1	Advanced SIMD copy	-
0xx0	01	00xx	xxx0xxxx1	UNALLOCATED	-
0xx0	0x	0111	00xxxxx10	UNALLOCATED	-
0xx0	0x	10xx	xxx00xxx1	Advanced SIMD three same (FP16)	-
0xx0	0x	10xx	xxx01xxx1	UNALLOCATED	-
0xx0	0x	1111	00xxxxx10	Advanced SIMD two-register miscellaneous (FP16)	-
0xx0	0x	x0xx	xxx1xxxx0	UNALLOCATED	-
0xx0	0x	x0xx	xxx1xxxx1	Advanced SIMD three-register extension	-
0xx0	0x	x100	00xxxxx10	Advanced SIMD two-register miscellaneous	-
0xx0	0x	x110	00xxxxx10	Advanced SIMD across lanes	-
0xx0	0x	x1xx	1xxxxxx10	UNALLOCATED	-
0xx0	0x	x1xx	x1xxxxx10	UNALLOCATED	-
0xx0	0x	x1xx	xxxxxxx00	Advanced SIMD three different	-
0xx0	0x	x1xx	xxxxxxxx1	Advanced SIMD three same	-
0xx0	10	0000	xxxxxxxx1	Advanced SIMD modified immediate	-

0xx0	10	!= 0000	xxxxxxxx1	Advanced SIMD shift by immediate	-
0xx0	11		xxxxxxxx1	UNALLOCATED	-
0xx0	1x		xxxxxxxx0	Advanced SIMD vector x indexed element	-
1100	00	10xx	xxx10xxxx	Cryptographic three-register, imm2	-
1100	00	11xx	xxx1x00xx	Cryptographic three-register SHA 512	-
1100	00		xxx0xxxxx	Cryptographic four-register	-
1100	01	00xx		XAR	FEAT_SHA3
1100	01	1000	0001000xx	Cryptographic two-register SHA 512	-
1xx0	1x			UNALLOCATED	-
x0x1	0x	x0xx		Conversion between floating-point and fixed-point	-
x0x1	0x	x1xx	xxx000000	Conversion between floating-point and integer	-
x0x1	0x	x1xx	xxxx10000	Floating-point data-processing (1 source)	-
x0x1	0x	x1xx	xxxxx1000	Floating-point compare	-
x0x1	0x	x1xx	xxxxxx100	Floating-point immediate	-
x0x1	0x	x1xx	xxxxxxx01	Floating-point conditional compare	-
x0x1	0x	x1xx	xxxxxxx10	Floating-point data-processing (2 source)	-
x0x1	0x	x1xx	xxxxxxx11	Floating-point conditional select	-
x0x1	1x			Floating-point data-processing (3 source)	-

Cryptographic AES

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	size		1	0	1	0	0	opcode					1	0	Rn					Rd				

Decode fields size	opcode	Instruction Details	Feature
	x1xxx	UNALLOCATED	-
	000xx	UNALLOCATED	-
	1xxxx	UNALLOCATED	-
x1		UNALLOCATED	-
00	00100	AESE	FEAT_AES
00	00101	AESD	FEAT_AES
00	00110	AESMC	-
00	00111	AESIMC	-
1x		UNALLOCATED	-

Cryptographic three-register SHA

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size		0	Rm					0	opcode			0	0	Rn					Rd				

Decode fields size	opcode	Instruction Details	Feature
	111	UNALLOCATED	-
x1		UNALLOCATED	-
00	000	SHA1C	FEAT_SHA1

Decode fields size	opcode	Instruction Details	Feature
00	001	SHA1P	FEAT_SHA1
00	010	SHA1M	FEAT_SHA1
00	011	SHA1SU0	FEAT_SHA1
00	100	SHA256H	FEAT_SHA256
00	101	SHA256H2	FEAT_SHA256
00	110	SHA256SU1	FEAT_SHA256
1x		UNALLOCATED	-

Cryptographic two-register SHA

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	0	size		1	0	1	0	0	opcode					1	0	Rn					Rd				

Decode fields size	opcode	Instruction Details	Feature
	xx1xx	UNALLOCATED	-
	x1xxx	UNALLOCATED	-
	1xxxx	UNALLOCATED	-
x1		UNALLOCATED	-
00	00000	SHA1H	FEAT_SHA1
00	00001	SHA1SU1	FEAT_SHA1
00	00010	SHA256SU0	FEAT_SHA256
00	00011	UNALLOCATED	-
1x		UNALLOCATED	-

Advanced SIMD scalar copy

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	op	1	1	1	1	0	0	0	0	imm5					0	imm4				1	Rn					Rd				

Decode fields op	imm4	Instruction Details
0	xxx1	UNALLOCATED
0	xx1x	UNALLOCATED
0	x1xx	UNALLOCATED
0	0000	DUP (element)
0	1xxx	UNALLOCATED
1		UNALLOCATED

Advanced SIMD scalar three same FP16

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	U	1	1	1	1	0	a	1	0	Rm					0	0	opcode			1	Rn					Rd				

Decode fields U	a	opcode	Instruction Details	Feature
		110	UNALLOCATED	-
	1	011	UNALLOCATED	-
0	0	011	FMULX	FEAT_FP16
0	0	100	FCMEQ (register)	FEAT_FP16
0	0	101	UNALLOCATED	-
0	0	111	FRECPS	FEAT_FP16
0	1	100	UNALLOCATED	-
0	1	101	UNALLOCATED	-
0	1	111	FRSQRTS	FEAT_FP16
1	0	011	UNALLOCATED	-
1	0	100	FCMGE (register)	FEAT_FP16
1	0	101	FACGE	FEAT_FP16
1	0	111	UNALLOCATED	-
1	1	010	FABD	FEAT_FP16
1	1	100	FCMGT (register)	FEAT_FP16
1	1	101	FACGT	FEAT_FP16
1	1	111	UNALLOCATED	-

Advanced SIMD scalar two-register miscellaneous FP16

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

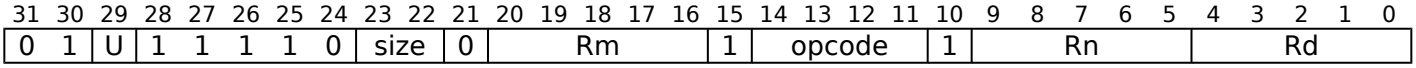
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	U	1	1	1	1	0	a	1	1	1	1	0	0	opcode					1	0	Rn					Rd				

Decode fields U	a	opcode	Instruction Details	Feature
		00xxx	UNALLOCATED	-
		010xx	UNALLOCATED	-
		10xxx	UNALLOCATED	-
		1100x	UNALLOCATED	-
		11110	UNALLOCATED	-
	0	011xx	UNALLOCATED	-
	0	11111	UNALLOCATED	-
	1	01111	UNALLOCATED	-
	1	11100	UNALLOCATED	-
0	0	11010	FCVTNS (vector)	FEAT_FP16
0	0	11011	FCVTMS (vector)	FEAT_FP16
0	0	11100	FCVTAS (vector)	FEAT_FP16
0	0	11101	SCVTF (vector, integer)	FEAT_FP16
0	1	01100	FCMGT (zero)	FEAT_FP16
0	1	01101	FCMEQ (zero)	FEAT_FP16
0	1	01110	FCMLT (zero)	FEAT_FP16
0	1	11010	FCVTPS (vector)	FEAT_FP16
0	1	11011	FCVTZS (vector, integer)	FEAT_FP16
0	1	11101	FRECPE	FEAT_FP16
0	1	11111	FRECPX	FEAT_FP16
1	0	11010	FCVTNU (vector)	FEAT_FP16
1	0	11011	FCVTMU (vector)	FEAT_FP16

Decode fields U	a	opcode	Instruction Details	Feature
1	0	11100	FCVTAU (vector)	FEAT_FP16
1	0	11101	UCVTF (vector, integer)	FEAT_FP16
1	1	01100	FCMGE (zero)	FEAT_FP16
1	1	01101	FCMLE (zero)	FEAT_FP16
1	1	01110	UNALLOCATED	-
1	1	11010	FCVTPU (vector)	FEAT_FP16
1	1	11011	FCVTZU (vector, integer)	FEAT_FP16
1	1	11101	FRSQRTE	FEAT_FP16
1	1	11111	UNALLOCATED	-

Advanced SIMD scalar three same extra

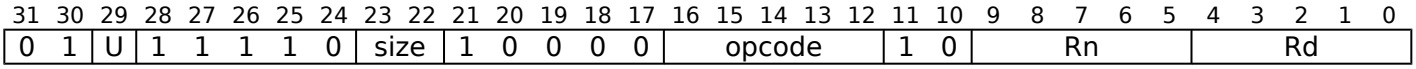
These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.



Decode fields U	opcode	Instruction Details	Feature
	001x	UNALLOCATED	-
	01xx	UNALLOCATED	-
	1xxx	UNALLOCATED	-
0	0000	UNALLOCATED	-
0	0001	UNALLOCATED	-
1	0000	SQRDMLAH (vector)	FEAT_RDM
1	0001	SQRDMLSH (vector)	FEAT_RDM

Advanced SIMD scalar two-register miscellaneous

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

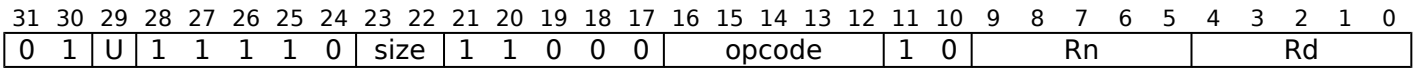


Decode fields U	size	opcode	Instruction Details
		0000x	UNALLOCATED
		00010	UNALLOCATED
		0010x	UNALLOCATED
		00110	UNALLOCATED
		01111	UNALLOCATED
		1000x	UNALLOCATED
		10011	UNALLOCATED
		10101	UNALLOCATED
		10111	UNALLOCATED
		1100x	UNALLOCATED
		11110	UNALLOCATED
	0x	011xx	UNALLOCATED
	0x	11111	UNALLOCATED
	1x	10110	UNALLOCATED

Decode fields			Instruction Details
U	size	opcode	
	1x	11100	UNALLOCATED
0		00011	SUQADD
0		00111	SQABS
0		01000	CMGT (zero)
0		01001	CMEQ (zero)
0		01010	CMLT (zero)
0		01011	ABS
0		10010	UNALLOCATED
0		10100	SQXTN, SQXTN2
0	0x	10110	UNALLOCATED
0	0x	11010	FCVTNS (vector)
0	0x	11011	FCVTMS (vector)
0	0x	11100	FCVTAS (vector)
0	0x	11101	SCVTF (vector, integer)
0	1x	01100	FCMGT (zero)
0	1x	01101	FCMEQ (zero)
0	1x	01110	FCMLT (zero)
0	1x	11010	FCVTPS (vector)
0	1x	11011	FCVTZS (vector, integer)
0	1x	11101	FRECPE
0	1x	11111	FRECPX
1		00011	USQADD
1		00111	SQNEG
1		01000	CMGE (zero)
1		01001	CMLE (zero)
1		01010	UNALLOCATED
1		01011	NEG (vector)
1		10010	SQXTUN, SQXTUN2
1		10100	UQXTN, UQXTN2
1	0x	10110	FCVTXN, FCVTXN2
1	0x	11010	FCVTNU (vector)
1	0x	11011	FCVTMU (vector)
1	0x	11100	FCVTAU (vector)
1	0x	11101	UCVTF (vector, integer)
1	1x	01100	FCMGE (zero)
1	1x	01101	FCMLE (zero)
1	1x	01110	UNALLOCATED
1	1x	11010	FCVTPU (vector)
1	1x	11011	FCVTZU (vector, integer)
1	1x	11101	FRSQRTE
1	1x	11111	UNALLOCATED

Advanced SIMD scalar pairwise

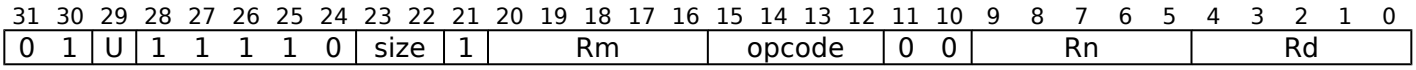
These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.



Decode fields U	size	opcode	Instruction Details	Feature
		00xxx	UNALLOCATED	-
		010xx	UNALLOCATED	-
		01110	UNALLOCATED	-
		10xxx	UNALLOCATED	-
		1100x	UNALLOCATED	-
		11010	UNALLOCATED	-
		111xx	UNALLOCATED	-
	1x	01101	UNALLOCATED	-
0		11011	ADDP (scalar)	-
0	0x	01100	FMAXNMP (scalar) — half-precision	FEAT_FP16
0	0x	01101	FADDP (scalar) — half-precision	FEAT_FP16
0	0x	01111	FMAXP (scalar) — half-precision	FEAT_FP16
0	1x	01100	FMINNMP (scalar) — half-precision	FEAT_FP16
0	1x	01111	FMINP (scalar) — half-precision	FEAT_FP16
1		11011	UNALLOCATED	-
1	0x	01100	FMAXNMP (scalar) — single-precision and double-precision	-
1	0x	01101	FADDP (scalar) — single-precision and double-precision	-
1	0x	01111	FMAXP (scalar) — single-precision and double-precision	-
1	1x	01100	FMINNMP (scalar) — single-precision and double-precision	-
1	1x	01111	FMINP (scalar) — single-precision and double-precision	-

Advanced SIMD scalar three different

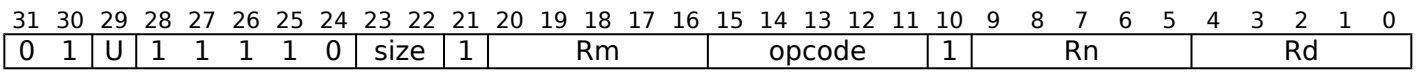
These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.



Decode fields U	opcode	Instruction Details
	00xx	UNALLOCATED
	01xx	UNALLOCATED
	1000	UNALLOCATED
	1010	UNALLOCATED
	1100	UNALLOCATED
	111x	UNALLOCATED
0	1001	SQDMLAL, SQDMLAL2 (vector)
0	1011	SQDMLSL, SQDMLSL2 (vector)
0	1101	SQDMULL, SQDMULL2 (vector)
1	1001	UNALLOCATED
1	1011	UNALLOCATED
1	1101	UNALLOCATED

Advanced SIMD scalar three same

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

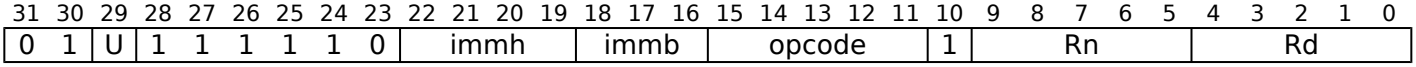


Decode fields			Instruction Details
U	size	opcode	
		00000	UNALLOCATED
		0001x	UNALLOCATED
		00100	UNALLOCATED
		011xx	UNALLOCATED
		1001x	UNALLOCATED
	1x	11011	UNALLOCATED
0		00001	SQADD
0		00101	SQSUB
0		00110	CMGT (register)
0		00111	CMGE (register)
0		01000	SSHL
0		01001	SQSHL (register)
0		01010	SRSHL
0		01011	SQRSHL
0		10000	ADD (vector)
0		10001	CMTST
0		10100	UNALLOCATED
0		10101	UNALLOCATED
0		10110	SQDMULH (vector)
0		10111	UNALLOCATED
0	0x	11000	UNALLOCATED
0	0x	11001	UNALLOCATED
0	0x	11010	UNALLOCATED
0	0x	11011	FMULX
0	0x	11100	FCMEQ (register)
0	0x	11101	UNALLOCATED
0	0x	11110	UNALLOCATED
0	0x	11111	FRECPS
0	1x	11000	UNALLOCATED
0	1x	11001	UNALLOCATED
0	1x	11010	UNALLOCATED
0	1x	11100	UNALLOCATED
0	1x	11101	UNALLOCATED
0	1x	11110	UNALLOCATED
0	1x	11111	FRSQRTS
1		00001	UQADD
1		00101	UQSUB
1		00110	CMHI (register)
1		00111	CMHS (register)
1		01000	USHL
1		01001	UQSHL (register)
1		01010	URSHL
1		01011	UQRSHL
1		10000	SUB (vector)
1		10001	CMEQ (register)
1		10100	UNALLOCATED
1		10101	UNALLOCATED

Decode fields U	size	opcode	Instruction Details
1		10110	SQRDMULH (vector)
1		10111	UNALLOCATED
1	0x	11000	UNALLOCATED
1	0x	11001	UNALLOCATED
1	0x	11010	UNALLOCATED
1	0x	11011	UNALLOCATED
1	0x	11100	FCMGE (register)
1	0x	11101	FACGE
1	0x	11110	UNALLOCATED
1	0x	11111	UNALLOCATED
1	1x	11000	UNALLOCATED
1	1x	11001	UNALLOCATED
1	1x	11010	FABD
1	1x	11100	FCMGT (register)
1	1x	11101	FACGT
1	1x	11110	UNALLOCATED
1	1x	11111	UNALLOCATED

Advanced SIMD scalar shift by immediate

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.



U	Decode fields immh	opcode	Instruction Details
	!= 0000	00001	UNALLOCATED
	!= 0000	00011	UNALLOCATED
	!= 0000	00101	UNALLOCATED
	!= 0000	00111	UNALLOCATED
	!= 0000	01001	UNALLOCATED
	!= 0000	01011	UNALLOCATED
	!= 0000	01101	UNALLOCATED
	!= 0000	01111	UNALLOCATED
	!= 0000	101xx	UNALLOCATED
	!= 0000	110xx	UNALLOCATED
	!= 0000	11101	UNALLOCATED
	!= 0000	11110	UNALLOCATED
	0000		UNALLOCATED
0	!= 0000	00000	SSHR
0	!= 0000	00010	SSRA
0	!= 0000	00100	SRSHR
0	!= 0000	00110	SRSRA
0	!= 0000	01000	UNALLOCATED
0	!= 0000	01010	SHL
0	!= 0000	01100	UNALLOCATED
0	!= 0000	01110	SQSHL (immediate)
0	!= 0000	10000	UNALLOCATED

Decode fields U	immh	opcode	Instruction Details
0	!= 0000	10001	UNALLOCATED
0	!= 0000	10010	SQSHRN, SQSHRN2
0	!= 0000	10011	SQRSHRN, SQRSHRN2
0	!= 0000	11100	SCVTF (vector, fixed-point)
0	!= 0000	11111	FCVTZS (vector, fixed-point)
1	!= 0000	00000	USHR
1	!= 0000	00010	USRA
1	!= 0000	00100	URSHR
1	!= 0000	00110	URSRA
1	!= 0000	01000	SRI
1	!= 0000	01010	SLI
1	!= 0000	01100	SQSHLU
1	!= 0000	01110	UQSHL (immediate)
1	!= 0000	10000	SQSHRUN, SQSHRUN2
1	!= 0000	10001	SQRSHRUN, SQRSHRUN2
1	!= 0000	10010	UQSHRN, UQSHRN2
1	!= 0000	10011	UQRSHRN, UQRSHRN2
1	!= 0000	11100	UCVTF (vector, fixed-point)
1	!= 0000	11111	FCVTZU (vector, fixed-point)

Advanced SIMD scalar x indexed element

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	U	1	1	1	1	1	size		L	M	Rm				opcode				H	0	Rn					Rd				

Decode fields U	size	opcode	Instruction Details	Feature
		0000	UNALLOCATED	-
		0010	UNALLOCATED	-
		0100	UNALLOCATED	-
		0110	UNALLOCATED	-
		1000	UNALLOCATED	-
		1010	UNALLOCATED	-
		1110	UNALLOCATED	-
	01	0001	UNALLOCATED	-
	01	0101	UNALLOCATED	-
	01	1001	UNALLOCATED	-
0		0011	SQDMLAL, SQDMLAL2 (by element)	-
0		0111	SQDMLSL, SQDMLSL2 (by element)	-
0		1011	SQDMULL, SQDMULL2 (by element)	-
0		1100	SQDMULH (by element)	-
0		1101	SQRDMULH (by element)	-
0		1111	UNALLOCATED	-
0	00	0001	FMLA (by element) — half-precision	FEAT_FP16
0	00	0101	FMLS (by element) — half-precision	FEAT_FP16
0	00	1001	FMUL (by element) — half-precision	FEAT_FP16
0	1x	0001	FMLA (by element) — single-precision and double-precision	-

Decode fields U	size	opcode	Instruction Details	Feature
0	1x	0101	FMLS (by element) — single-precision and double-precision	-
0	1x	1001	FMUL (by element) — single-precision and double-precision	-
1		0011	UNALLOCATED	-
1		0111	UNALLOCATED	-
1		1011	UNALLOCATED	-
1		1100	UNALLOCATED	-
1		1101	SQRDMLAH (by element)	FEAT_RDM
1		1111	SQRDMLSH (by element)	FEAT_RDM
1	00	0001	UNALLOCATED	-
1	00	0101	UNALLOCATED	-
1	00	1001	FMULX (by element) — half-precision	FEAT_FP16
1	1x	0001	UNALLOCATED	-
1	1x	0101	UNALLOCATED	-
1	1x	1001	FMULX (by element) — single-precision and double-precision	-

Advanced SIMD table lookup

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	op2		0	Rm					0	len		op	0	0	Rn					Rd				

Decode fields op2	len	op	Instruction Details
x1			UNALLOCATED
00	00	0	TBL — single register table
00	00	1	TBX — single register table
00	01	0	TBL — two register table
00	01	1	TBX — two register table
00	10	0	TBL — three register table
00	10	1	TBX — three register table
00	11	0	TBL — four register table
00	11	1	TBX — four register table
1x			UNALLOCATED

Advanced SIMD permute

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	0	0	1	1	1	0	size		0	Rm					0	opcode			1	0	Rn					Rd				

Decode fields opcode	Instruction Details
000	UNALLOCATED
001	UZP1
010	TRN1
011	ZIP1
100	UNALLOCATED
101	UZP2

Decode fields opcode	Instruction Details
110	TRN2
111	ZIP2

Advanced SIMD extract

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	1	0	1	1	1	0	op2		0	Rm					0	imm4				0	Rn					Rd				

Decode fields op2	Instruction Details
x1	UNALLOCATED
00	EXT
1x	UNALLOCATED

Advanced SIMD copy

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	op	0	1	1	1	0	0	0	0	imm5					0	imm4				1	Rn					Rd				

Decode fields Q	op	imm5	imm4	Instruction Details
		x0000		UNALLOCATED
	0		0000	DUP (element)
	0		0001	DUP (general)
	0		0010	UNALLOCATED
	0		0100	UNALLOCATED
	0		0110	UNALLOCATED
	0		1xxx	UNALLOCATED
0	0		0011	UNALLOCATED
0	0		0101	SMOV
0	0		0111	UMOV
0	1			UNALLOCATED
1	0		0011	INS (general)
1	0		0101	SMOV
1	0	x1000	0111	UMOV
1	1			INS (element)

Advanced SIMD three same (FP16)

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

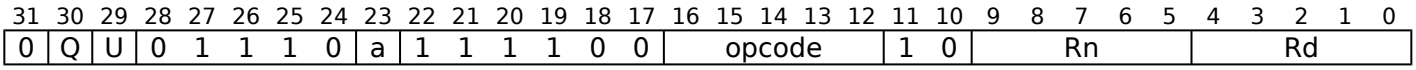
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	U	0	1	1	1	0	a	1	0	Rm					0	0	opcode			1	Rn					Rd				

Decode fields U	a	opcode	Instruction Details	Feature
0	0	000	FMAXNM (vector)	FEAT_FP16
0	0	001	FMLA (vector)	FEAT_FP16

Decode fields U	a	opcode	Instruction Details	Feature
0	0	010	FADD (vector)	FEAT_FP16
0	0	011	FMULX	FEAT_FP16
0	0	100	FCMEQ (register)	FEAT_FP16
0	0	101	UNALLOCATED	-
0	0	110	FMAX (vector)	FEAT_FP16
0	0	111	FRECPS	FEAT_FP16
0	1	000	FMINNM (vector)	FEAT_FP16
0	1	001	FMLS (vector)	FEAT_FP16
0	1	010	FSUB (vector)	FEAT_FP16
0	1	011	UNALLOCATED	-
0	1	100	UNALLOCATED	-
0	1	101	UNALLOCATED	-
0	1	110	FMIN (vector)	FEAT_FP16
0	1	111	FRSQRTS	FEAT_FP16
1	0	000	FMAXNMP (vector)	FEAT_FP16
1	0	001	UNALLOCATED	-
1	0	010	FADDP (vector)	FEAT_FP16
1	0	011	FMUL (vector)	FEAT_FP16
1	0	100	FCMGE (register)	FEAT_FP16
1	0	101	FACGE	FEAT_FP16
1	0	110	FMAXP (vector)	FEAT_FP16
1	0	111	FDIV (vector)	FEAT_FP16
1	1	000	FMINNMP (vector)	FEAT_FP16
1	1	001	UNALLOCATED	-
1	1	010	FABD	FEAT_FP16
1	1	011	UNALLOCATED	-
1	1	100	FCMGT (register)	FEAT_FP16
1	1	101	FACGT	FEAT_FP16
1	1	110	FMINP (vector)	FEAT_FP16
1	1	111	UNALLOCATED	-

Advanced SIMD two-register miscellaneous (FP16)

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

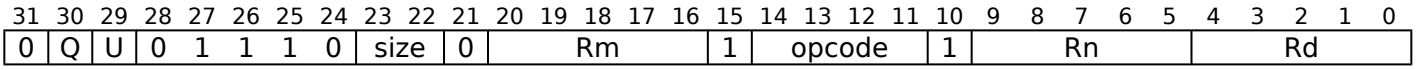


Decode fields U	a	opcode	Instruction Details	Feature
		00xxx	UNALLOCATED	-
		010xx	UNALLOCATED	-
		10xxx	UNALLOCATED	-
		11110	UNALLOCATED	-
	0	011xx	UNALLOCATED	-
	0	11111	UNALLOCATED	-
	1	11100	UNALLOCATED	-
0	0	11000	FRINTN (vector)	FEAT_FP16
0	0	11001	FRINTM (vector)	FEAT_FP16

Decode fields			Instruction Details	Feature
U	a	opcode		
0	0	11010	FCVTNS (vector)	FEAT_FP16
0	0	11011	FCVTMS (vector)	FEAT_FP16
0	0	11100	FCVTAS (vector)	FEAT_FP16
0	0	11101	SCVTF (vector, integer)	FEAT_FP16
0	1	01100	FCMGT (zero)	FEAT_FP16
0	1	01101	FCMEQ (zero)	FEAT_FP16
0	1	01110	FCMLT (zero)	FEAT_FP16
0	1	01111	FABS (vector)	FEAT_FP16
0	1	11000	FRINTP (vector)	FEAT_FP16
0	1	11001	FRINTZ (vector)	FEAT_FP16
0	1	11010	FCVTPS (vector)	FEAT_FP16
0	1	11011	FCVTZS (vector, integer)	FEAT_FP16
0	1	11101	FRECPE	FEAT_FP16
0	1	11111	UNALLOCATED	-
1	0	11000	FRINTA (vector)	FEAT_FP16
1	0	11001	FRINTX (vector)	FEAT_FP16
1	0	11010	FCVTNU (vector)	FEAT_FP16
1	0	11011	FCVTMU (vector)	FEAT_FP16
1	0	11100	FCVTAU (vector)	FEAT_FP16
1	0	11101	UCVTF (vector, integer)	FEAT_FP16
1	1	01100	FCMGE (zero)	FEAT_FP16
1	1	01101	FCMLE (zero)	FEAT_FP16
1	1	01110	UNALLOCATED	-
1	1	01111	FNEG (vector)	FEAT_FP16
1	1	11000	UNALLOCATED	-
1	1	11001	FRINTI (vector)	FEAT_FP16
1	1	11010	FCVTPU (vector)	FEAT_FP16
1	1	11011	FCVTZU (vector, integer)	FEAT_FP16
1	1	11101	FRSQRTE	FEAT_FP16
1	1	11111	FSQRT (vector)	FEAT_FP16

Advanced SIMD three-register extension

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

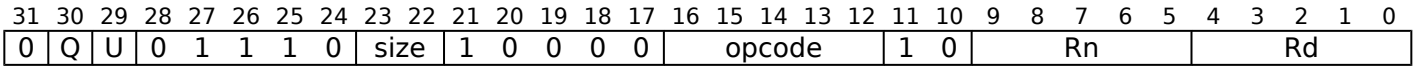


Decode fields				Instruction Details	Feature
Q	U	size	opcode		
		0x	0011	UNALLOCATED	-
		11	0011	UNALLOCATED	-
	0		0000	UNALLOCATED	-
	0		0001	UNALLOCATED	-
	0		0010	SDOT (vector)	FEAT_DotProd
	0		1xxx	UNALLOCATED	-
	0	10	0011	USDOT (vector)	FEAT_I8MM
	1		0000	SQRDMLAH (vector)	FEAT_RDM
	1		0001	SQRDMLSH (vector)	FEAT_RDM

Decode fields				Instruction Details	Feature
Q	U	size	opcode		
	1		0010	UDOT (vector)	FEAT_DotProd
	1		10xx	FCMLA	FEAT_FCMA
	1		11x0	FCADD	FEAT_FCMA
	1	00	1101	UNALLOCATED	-
	1	00	1111	UNALLOCATED	-
	1	01	1111	BFDOT (vector)	FEAT_BF16
	1	1x	1101	UNALLOCATED	-
	1	10	0011	UNALLOCATED	-
	1	10	1111	UNALLOCATED	-
	1	11	1111	BFMLALB, BFMLALT (vector)	FEAT_BF16
0			01xx	UNALLOCATED	-
0	1	01	1101	UNALLOCATED	-
1		0x	01xx	UNALLOCATED	-
1		1x	011x	UNALLOCATED	-
1	0	10	0100	SMMLA (vector)	FEAT_I8MM
1	0	10	0101	USMMLA (vector)	FEAT_I8MM
1	1	01	1101	BFMMLA	FEAT_BF16
1	1	10	0100	UMMLA (vector)	FEAT_I8MM
1	1	10	0101	UNALLOCATED	-

Advanced SIMD two-register miscellaneous

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.



Decode fields			Instruction Details	Feature
U	size	opcode		
		1000x	UNALLOCATED	-
		10101	UNALLOCATED	-
	0x	011xx	UNALLOCATED	-
	1x	10111	UNALLOCATED	-
	1x	11110	UNALLOCATED	-
	11	10110	UNALLOCATED	-
0		00000	REV64	-
0		00001	REV16 (vector)	-
0		00010	SADDLP	-
0		00011	SUQADD	-
0		00100	CLS (vector)	-
0		00101	CNT	-
0		00110	SADALP	-
0		00111	SQABS	-
0		01000	CMGT (zero)	-
0		01001	CMEQ (zero)	-
0		01010	CMLT (zero)	-
0		01011	ABS	-
0		10010	XTN, XTN2	-
0		10011	UNALLOCATED	-

Decode fields			Instruction Details	Feature
U	size	opcode		
0		10100	SQXTN, SQXTN2	-
0	0x	10110	FCVTN, FCVTN2	-
0	0x	10111	FCVTL, FCVTL2	-
0	0x	11000	FRINTN (vector)	-
0	0x	11001	FRINTM (vector)	-
0	0x	11010	FCVTNS (vector)	-
0	0x	11011	FCVTMS (vector)	-
0	0x	11100	FCVTAS (vector)	-
0	0x	11101	SCVTF (vector, integer)	-
0	0x	11110	FRINT32Z (vector)	FEAT_FRINTTS
0	0x	11111	FRINT64Z (vector)	FEAT_FRINTTS
0	1x	01100	FCMGT (zero)	-
0	1x	01101	FCMEQ (zero)	-
0	1x	01110	FCMLT (zero)	-
0	1x	01111	FABS (vector)	-
0	1x	11000	FRINTP (vector)	-
0	1x	11001	FRINTZ (vector)	-
0	1x	11010	FCVTPS (vector)	-
0	1x	11011	FCVTZS (vector, integer)	-
0	1x	11100	URECPE	-
0	1x	11101	FRECPE	-
0	1x	11111	UNALLOCATED	-
0	10	10110	BFCVTN, BFCVTN2	FEAT_BF16
1		00000	REV32 (vector)	-
1		00001	UNALLOCATED	-
1		00010	UADDLP	-
1		00011	USQADD	-
1		00100	CLZ (vector)	-
1		00110	UADALP	-
1		00111	SQNEG	-
1		01000	CMGE (zero)	-
1		01001	CMLE (zero)	-
1		01010	UNALLOCATED	-
1		01011	NEG (vector)	-
1		10010	SQXTUN, SQXTUN2	-
1		10011	SHLL, SHLL2	-
1		10100	UQXTN, UQXTN2	-
1	0x	10110	FCVTXN, FCVTXN2	-
1	0x	10111	UNALLOCATED	-
1	0x	11000	FRINTA (vector)	-
1	0x	11001	FRINTX (vector)	-
1	0x	11010	FCVTNU (vector)	-
1	0x	11011	FCVTMU (vector)	-
1	0x	11100	FCVTAU (vector)	-
1	0x	11101	UCVTF (vector, integer)	-
1	0x	11110	FRINT32X (vector)	FEAT_FRINTTS
1	0x	11111	FRINT64X (vector)	FEAT_FRINTTS

Decode fields			Instruction Details	Feature
U	size	opcode		
1	00	00101	NOT	-
1	01	00101	RBIT (vector)	-
1	1x	00101	UNALLOCATED	-
1	1x	01100	FCMGE (zero)	-
1	1x	01101	FCMLE (zero)	-
1	1x	01110	UNALLOCATED	-
1	1x	01111	FNEG (vector)	-
1	1x	11000	UNALLOCATED	-
1	1x	11001	FRINTI (vector)	-
1	1x	11010	FCVTPU (vector)	-
1	1x	11011	FCVTZU (vector, integer)	-
1	1x	11100	URSQRTE	-
1	1x	11101	FRSQRTE	-
1	1x	11111	FSQRT (vector)	-
1	10	10110	UNALLOCATED	-

Advanced SIMD across lanes

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	U	0	1	1	1	0	size		1	1	0	0	0	opcode					1	0	Rn					Rd				

Decode fields			Instruction Details	Feature
U	size	opcode		
		0000x	UNALLOCATED	-
		00010	UNALLOCATED	-
		001xx	UNALLOCATED	-
		0100x	UNALLOCATED	-
		01011	UNALLOCATED	-
		01101	UNALLOCATED	-
		01110	UNALLOCATED	-
		10xxx	UNALLOCATED	-
		1100x	UNALLOCATED	-
		111xx	UNALLOCATED	-
0		00011	SADDLV	-
0		01010	SMAXV	-
0		11010	SMINV	-
0		11011	ADDV	-
0	00	01100	FMAXNMV — half-precision	FEAT_FP16
0	00	01111	FMAXV — half-precision	FEAT_FP16
0	01	01100	UNALLOCATED	-
0	01	01111	UNALLOCATED	-
0	10	01100	FMINNMV — half-precision	FEAT_FP16
0	10	01111	FMINV — half-precision	FEAT_FP16
0	11	01100	UNALLOCATED	-
0	11	01111	UNALLOCATED	-
1		00011	UADDLV	-
1		01010	UMAXV	-

Decode fields U	size	opcode	Instruction Details	Feature
1		11010	UMINV	-
1		11011	UNALLOCATED	-
1	0x	01100	FMAXNMV — single-precision and double-precision	-
1	0x	01111	FMAXV — single-precision and double-precision	-
1	1x	01100	FMINNMV — single-precision and double-precision	-
1	1x	01111	FMINV — single-precision and double-precision	-

Advanced SIMD three different

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	U	0	1	1	1	0	size		1	Rm					opcode				0	0	Rn					Rd				

Decode fields U	opcode	Instruction Details
	1111	UNALLOCATED
0	0000	SADDL, SADDL2
0	0001	SADDW, SADDW2
0	0010	SSUBL, SSUBL2
0	0011	SSUBW, SSUBW2
0	0100	ADDHN, ADDHN2
0	0101	SABAL, SABAL2
0	0110	SUBHN, SUBHN2
0	0111	SABDL, SABDL2
0	1000	SMLAL, SMLAL2 (vector)
0	1001	SQDMLAL, SQDMLAL2 (vector)
0	1010	SMLSL, SMLSL2 (vector)
0	1011	SQDMLSL, SQDMLSL2 (vector)
0	1100	SMULL, SMULL2 (vector)
0	1101	SQDMULL, SQDMULL2 (vector)
0	1110	PMULL, PMULL2
1	0000	UADDL, UADDL2
1	0001	UADDW, UADDW2
1	0010	USUBL, USUBL2
1	0011	USUBW, USUBW2
1	0100	RADDHN, RADDHN2
1	0101	UABAL, UABAL2
1	0110	RSUBHN, RSUBHN2
1	0111	UABDL, UABDL2
1	1000	UMLAL, UMLAL2 (vector)
1	1001	UNALLOCATED
1	1010	UMLSL, UMLSL2 (vector)
1	1011	UNALLOCATED
1	1100	UMULL, UMULL2 (vector)
1	1101	UNALLOCATED
1	1110	UNALLOCATED

Advanced SIMD three same

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	U	0	1	1	1	0	size		1	Rm					opcode					1	Rn					Rd				

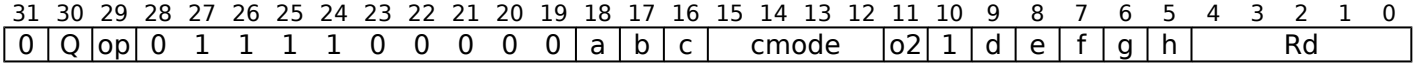
Decode fields			Instruction Details	Feature
U	size	opcode		
0		00000	SHADD	-
0		00001	SQADD	-
0		00010	SRHADD	-
0		00100	SHSUB	-
0		00101	SQSUB	-
0		00110	CMGT (register)	-
0		00111	CMGE (register)	-
0		01000	SSHL	-
0		01001	SQSHL (register)	-
0		01010	SRSHL	-
0		01011	SQRSHL	-
0		01100	SMAX	-
0		01101	SMIN	-
0		01110	SABD	-
0		01111	SABA	-
0		10000	ADD (vector)	-
0		10001	CMTST	-
0		10010	MLA (vector)	-
0		10011	MUL (vector)	-
0		10100	SMAXP	-
0		10101	SMINP	-
0		10110	SQDMULH (vector)	-
0		10111	ADDP (vector)	-
0	0x	11000	FMAXNM (vector)	-
0	0x	11001	FMLA (vector)	-
0	0x	11010	FADD (vector)	-
0	0x	11011	FMULX	-
0	0x	11100	FCMEQ (register)	-
0	0x	11110	FMAX (vector)	-
0	0x	11111	FRECPS	-
0	00	00011	AND (vector)	-
0	00	11101	FMLAL, FMLAL2 (vector) — FMLAL	FEAT_FHM
0	01	00011	BIC (vector, register)	-
0	01	11101	UNALLOCATED	-
0	1x	11000	FMINNM (vector)	-
0	1x	11001	FMLS (vector)	-
0	1x	11010	FSUB (vector)	-
0	1x	11011	UNALLOCATED	-
0	1x	11100	UNALLOCATED	-
0	1x	11110	FMIN (vector)	-
0	1x	11111	FRSQRTS	-
0	10	00011	ORR (vector, register)	-

Decode fields			Instruction Details	Feature
U	size	opcode		
0	10	11101	FMLSL, FMLSL2 (vector) — FMLSL	FEAT_FHM
0	11	00011	ORN (vector)	-
0	11	11101	UNALLOCATED	-
1		00000	UHADD	-
1		00001	UQADD	-
1		00010	URHADD	-
1		00100	UHSUB	-
1		00101	UQSUB	-
1		00110	CMHI (register)	-
1		00111	CMHS (register)	-
1		01000	USHL	-
1		01001	UQSHL (register)	-
1		01010	URSHL	-
1		01011	UQRSHL	-
1		01100	UMAX	-
1		01101	UMIN	-
1		01110	UABD	-
1		01111	UABA	-
1		10000	SUB (vector)	-
1		10001	CMEQ (register)	-
1		10010	MLS (vector)	-
1		10011	PMUL	-
1		10100	UMAXP	-
1		10101	UMINP	-
1		10110	SQRDMULH (vector)	-
1		10111	UNALLOCATED	-
1	0x	11000	FMAXNMP (vector)	-
1	0x	11010	FADDP (vector)	-
1	0x	11011	FMUL (vector)	-
1	0x	11100	FCMGE (register)	-
1	0x	11101	FACGE	-
1	0x	11110	FMAXP (vector)	-
1	0x	11111	FDIV (vector)	-
1	00	00011	EOR (vector)	-
1	00	11001	FMLAL, FMLAL2 (vector) — FMLAL2	FEAT_FHM
1	01	00011	BSL	-
1	01	11001	UNALLOCATED	-
1	1x	11000	FMINNMP (vector)	-
1	1x	11010	FABD	-
1	1x	11011	UNALLOCATED	-
1	1x	11100	FCMGT (register)	-
1	1x	11101	FACGT	-
1	1x	11110	FMINP (vector)	-
1	1x	11111	UNALLOCATED	-
1	10	00011	BIT	-
1	10	11001	FMLSL, FMLSL2 (vector) — FMLSL2	FEAT_FHM
1	11	00011	BIF	-

Decode fields U	size	opcode	Instruction Details	Feature
1	11	11001	UNALLOCATED	-

Advanced SIMD modified immediate

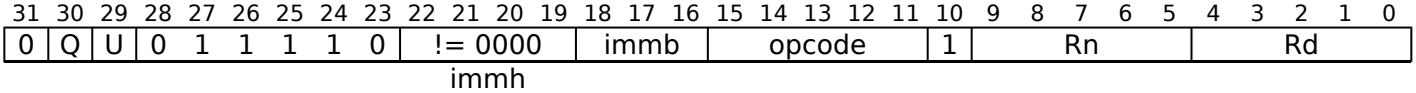
These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.



Q	Decode fields op	cmode	o2	Instruction Details	Feature
	0	0xxx	1	UNALLOCATED	-
	0	0xx0	0	MOVI — 32-bit shifted immediate	-
	0	0xx1	0	ORR (vector, immediate) — 32-bit	-
	0	10xx	1	UNALLOCATED	-
	0	10x0	0	MOVI — 16-bit shifted immediate	-
	0	10x1	0	ORR (vector, immediate) — 16-bit	-
	0	110x	0	MOVI — 32-bit shifting ones	-
	0	110x	1	UNALLOCATED	-
	0	1110	0	MOVI — 8-bit	-
	0	1110	1	UNALLOCATED	-
	0	1111	0	FMOV (vector, immediate) — single-precision	-
	0	1111	1	FMOV (vector, immediate) — half-precision	FEAT_FP16
	1		1	UNALLOCATED	-
	1	0xx0	0	MVNI — 32-bit shifted immediate	-
	1	0xx1	0	BIC (vector, immediate) — 32-bit	-
	1	10x0	0	MVNI — 16-bit shifted immediate	-
	1	10x1	0	BIC (vector, immediate) — 16-bit	-
	1	110x	0	MVNI — 32-bit shifting ones	-
0	1	1110	0	MOVI — 64-bit scalar	-
0	1	1111	0	UNALLOCATED	-
1	1	1110	0	MOVI — 64-bit vector	-
1	1	1111	0	FMOV (vector, immediate) — double-precision	-

Advanced SIMD shift by immediate

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.



The following constraints also apply to this encoding: immh != 0000 && immh != 0000

Decode fields U	opcode	Instruction Details
	00001	UNALLOCATED
	00011	UNALLOCATED
	00101	UNALLOCATED
	00111	UNALLOCATED
	01001	UNALLOCATED

Decode fields U	opcode	Instruction Details
	01011	UNALLOCATED
	01101	UNALLOCATED
	01111	UNALLOCATED
	10101	UNALLOCATED
	1011x	UNALLOCATED
	110xx	UNALLOCATED
	11101	UNALLOCATED
	11110	UNALLOCATED
0	00000	SSHR
0	00010	SSRA
0	00100	SRSHR
0	00110	SRSRA
0	01000	UNALLOCATED
0	01010	SHL
0	01100	UNALLOCATED
0	01110	SQSHL (immediate)
0	10000	SHRN, SHRN2
0	10001	RSHRN, RSHRN2
0	10010	SQSHRN, SQSHRN2
0	10011	SQRSHRN, SQRSHRN2
0	10100	SSHLL, SSHLL2
0	11100	SCVTF (vector, fixed-point)
0	11111	FCVTZS (vector, fixed-point)
1	00000	USHR
1	00010	USRA
1	00100	URSHR
1	00110	URSRA
1	01000	SRI
1	01010	SLI
1	01100	SQSHLU
1	01110	UQSHL (immediate)
1	10000	SQSHRUN, SQSHRUN2
1	10001	SQRSHRUN, SQRSHRUN2
1	10010	UQSHRN, UQSHRN2
1	10011	UQRSHRN, UQRSHRN2
1	10100	USHLL, USHLL2
1	11100	UCVTF (vector, fixed-point)
1	11111	FCVTZU (vector, fixed-point)

Advanced SIMD vector x indexed element

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Q	U	0	1	1	1	1	size		L	M	Rm				opcode				H	0	Rn					Rd				

Decode fields U	size	opcode	Instruction Details	Feature
	01	1001	UNALLOCATED	-

Decode fields			Instruction Details	Feature
U	size	opcode		
0		0010	SMLAL, SMLAL2 (by element)	-
0		0011	SQDMLAL, SQDMLAL2 (by element)	-
0		0110	SMLSL, SMLSL2 (by element)	-
0		0111	SQDMLSL, SQDMLSL2 (by element)	-
0		1000	MUL (by element)	-
0		1010	SMULL, SMULL2 (by element)	-
0		1011	SQDMULL, SQDMULL2 (by element)	-
0		1100	SQDMULH (by element)	-
0		1101	SQRDMULH (by element)	-
0		1110	SDOT (by element)	FEAT_DotProd
0	0x	0000	UNALLOCATED	-
0	0x	0100	UNALLOCATED	-
0	00	0001	FMLA (by element) — half-precision	FEAT_FP16
0	00	0101	FMLS (by element) — half-precision	FEAT_FP16
0	00	1001	FMUL (by element) — half-precision	FEAT_FP16
0	00	1111	SUDOT (by element)	FEAT_I8MM
0	01	0001	UNALLOCATED	-
0	01	0101	UNALLOCATED	-
0	01	1111	BFDOT (by element)	FEAT_BF16
0	1x	0001	FMLA (by element) — single-precision and double-precision	-
0	1x	0101	FMLS (by element) — single-precision and double-precision	-
0	1x	1001	FMUL (by element) — single-precision and double-precision	-
0	10	0000	FMLAL, FMLAL2 (by element) — FMLAL	FEAT_FHM
0	10	0100	FMLSL, FMLSL2 (by element) — FMLSL	FEAT_FHM
0	10	1111	USDOT (by element)	FEAT_I8MM
0	11	0000	UNALLOCATED	-
0	11	0100	UNALLOCATED	-
0	11	1111	BFMLALB, BFMLALT (by element)	FEAT_BF16
1		0000	MLA (by element)	-
1		0010	UMLAL, UMLAL2 (by element)	-
1		0100	MLS (by element)	-
1		0110	UMLSL, UMLSL2 (by element)	-
1		1010	UMULL, UMULL2 (by element)	-
1		1011	UNALLOCATED	-
1		1101	SQRDMLAH (by element)	FEAT_RDM
1		1110	UDOT (by element)	FEAT_DotProd
1		1111	SQRDMLSH (by element)	FEAT_RDM
1	0x	1000	UNALLOCATED	-
1	0x	1100	UNALLOCATED	-
1	00	0001	UNALLOCATED	-
1	00	0011	UNALLOCATED	-
1	00	0101	UNALLOCATED	-
1	00	0111	UNALLOCATED	-
1	00	1001	FMULX (by element) — half-precision	FEAT_FP16
1	01	0xx1	FCMLA (by element)	FEAT_FCMA
1	1x	1001	FMULX (by element) — single-precision and double-precision	-
1	10	0xx1	FCMLA (by element)	FEAT_FCMA

Decode fields U	size	opcode	Instruction Details	Feature
1	10	1000	FMLAL, FMLAL2 (by element) — FMLAL2	FEAT_FHM
1	10	1100	FMLSL, FMLSL2 (by element) — FMLSL2	FEAT_FHM
1	11	0001	UNALLOCATED	-
1	11	0011	UNALLOCATED	-
1	11	0101	UNALLOCATED	-
1	11	0111	UNALLOCATED	-
1	11	1000	UNALLOCATED	-
1	11	1100	UNALLOCATED	-

Cryptographic three-register, imm2

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	0	Rm					1	0	imm2		opcode		Rn					Rd				

Decode fields opcode	Instruction Details	Feature
00	SM3TT1A	FEAT_SM3
01	SM3TT1B	FEAT_SM3
10	SM3TT2A	FEAT_SM3
11	SM3TT2B	FEAT_SM3

Cryptographic three-register SHA 512

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	1	1	Rm					1	O	0	0	opcode		Rn					Rd				

Decode fields O	opcode	Instruction Details	Feature
0	00	SHA512H	FEAT_SHA512
0	01	SHA512H2	FEAT_SHA512
0	10	SHA512SU1	FEAT_SHA512
0	11	RAX1	FEAT_SHA3
1	00	SM3PARTW1	FEAT_SM3
1	01	SM3PARTW2	FEAT_SM3
1	10	SM4EKEY	FEAT_SM4
1	11	UNALLOCATED	-

Cryptographic four-register

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	0	Op0		Rm					0	Ra						Rn					Rd			

Decode fields Op0	Instruction Details	Feature
00	EOR3	FEAT_SHA3
01	BCAX	FEAT_SHA3

Decode fields Op0	Instruction Details	Feature
10	SM3SS1	FEAT_SM3
11	UNALLOCATED	-

Cryptographic two-register SHA 512

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	1	0	1	1	0	0	0	0	0	0	1	0	0	0	opcode		Rn					Rd				

Decode fields opcode	Instruction Details	Feature
00	SHA512SU0	FEAT_SHA512
01	SM4E	FEAT_SM4
1x	UNALLOCATED	-

Conversion between floating-point and fixed-point

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

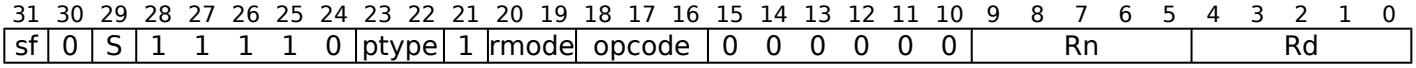
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
sf	0	S	1	1	1	1	0	ptype		0	rmode		opcode			scale						Rn					Rd				

Decode fields sf	S	ptype	rmode	opcode	scale	Instruction Details	Feature
				1xx		UNALLOCATED	-
			x0	00x		UNALLOCATED	-
			x1	01x		UNALLOCATED	-
			0x	00x		UNALLOCATED	-
			1x	01x		UNALLOCATED	-
		10				UNALLOCATED	-
	1					UNALLOCATED	-
0					0xxxxx	UNALLOCATED	-
0	0	00	00	010		SCVTF (scalar, fixed-point) — 32-bit to single-precision	-
0	0	00	00	011		UCVTF (scalar, fixed-point) — 32-bit to single-precision	-
0	0	00	11	000		FCVTZS (scalar, fixed-point) — single-precision to 32-bit	-
0	0	00	11	001		FCVTZU (scalar, fixed-point) — single-precision to 32-bit	-
0	0	01	00	010		SCVTF (scalar, fixed-point) — 32-bit to double-precision	-
0	0	01	00	011		UCVTF (scalar, fixed-point) — 32-bit to double-precision	-
0	0	01	11	000		FCVTZS (scalar, fixed-point) — double-precision to 32-bit	-
0	0	01	11	001		FCVTZU (scalar, fixed-point) — double-precision to 32-bit	-
0	0	11	00	010		SCVTF (scalar, fixed-point) — 32-bit to half-precision	FEAT_FP16
0	0	11	00	011		UCVTF (scalar, fixed-point) — 32-bit to half-precision	FEAT_FP16

Decode fields						Instruction Details	Feature
sf	S	ptype	rmode	opcode	scale		
0	0	11	11	000		FCVTZS (scalar, fixed-point) — half-precision to 32-bit	FEAT_FP16
0	0	11	11	001		FCVTZU (scalar, fixed-point) — half-precision to 32-bit	FEAT_FP16
1	0	00	00	010		SCVTF (scalar, fixed-point) — 64-bit to single-precision	-
1	0	00	00	011		UCVTF (scalar, fixed-point) — 64-bit to single-precision	-
1	0	00	11	000		FCVTZS (scalar, fixed-point) — single-precision to 64-bit	-
1	0	00	11	001		FCVTZU (scalar, fixed-point) — single-precision to 64-bit	-
1	0	01	00	010		SCVTF (scalar, fixed-point) — 64-bit to double-precision	-
1	0	01	00	011		UCVTF (scalar, fixed-point) — 64-bit to double-precision	-
1	0	01	11	000		FCVTZS (scalar, fixed-point) — double-precision to 64-bit	-
1	0	01	11	001		FCVTZU (scalar, fixed-point) — double-precision to 64-bit	-
1	0	11	00	010		SCVTF (scalar, fixed-point) — 64-bit to half-precision	FEAT_FP16
1	0	11	00	011		UCVTF (scalar, fixed-point) — 64-bit to half-precision	FEAT_FP16
1	0	11	11	000		FCVTZS (scalar, fixed-point) — half-precision to 64-bit	FEAT_FP16
1	0	11	11	001		FCVTZU (scalar, fixed-point) — half-precision to 64-bit	FEAT_FP16

Conversion between floating-point and integer

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.



Decode fields					Instruction Details	Feature
sf	S	ptype	rmode	opcode		
			x1	01x	UNALLOCATED	-
			x1	10x	UNALLOCATED	-
			1x	01x	UNALLOCATED	-
			1x	10x	UNALLOCATED	-
	0	10		0xx	UNALLOCATED	-
	0	10		10x	UNALLOCATED	-
	1				UNALLOCATED	-
0	0	00	x1	11x	UNALLOCATED	-
0	0	00	00	000	FCVTNS (scalar) — single-precision to 32-bit	-
0	0	00	00	001	FCVTNU (scalar) — single-precision to 32-bit	-
0	0	00	00	010	SCVTF (scalar, integer) — 32-bit to single-precision	-
0	0	00	00	011	UCVTF (scalar, integer) — 32-bit to single-precision	-
0	0	00	00	100	FCVTAS (scalar) — single-precision to 32-bit	-
0	0	00	00	101	FCVTAU (scalar) — single-precision to 32-bit	-
0	0	00	00	110	FMOV (general) — single-precision to 32-bit	-

Decode fields					Instruction Details	Feature
sf	S	ptype	rmode	opcode		
0	0	00	00	111	FMOV (general) — 32-bit to single-precision	-
0	0	00	01	000	FCVTPS (scalar) — single-precision to 32-bit	-
0	0	00	01	001	FCVTPU (scalar) — single-precision to 32-bit	-
0	0	00	1x	11x	UNALLOCATED	-
0	0	00	10	000	FCVTMS (scalar) — single-precision to 32-bit	-
0	0	00	10	001	FCVTMU (scalar) — single-precision to 32-bit	-
0	0	00	11	000	FCVTZS (scalar, integer) — single-precision to 32-bit	-
0	0	00	11	001	FCVTZU (scalar, integer) — single-precision to 32-bit	-
0	0	01	0x	11x	UNALLOCATED	-
0	0	01	00	000	FCVTNS (scalar) — double-precision to 32-bit	-
0	0	01	00	001	FCVTNU (scalar) — double-precision to 32-bit	-
0	0	01	00	010	SCVTF (scalar, integer) — 32-bit to double-precision	-
0	0	01	00	011	UCVTF (scalar, integer) — 32-bit to double-precision	-
0	0	01	00	100	FCVTAS (scalar) — double-precision to 32-bit	-
0	0	01	00	101	FCVTAU (scalar) — double-precision to 32-bit	-
0	0	01	01	000	FCVTPS (scalar) — double-precision to 32-bit	-
0	0	01	01	001	FCVTPU (scalar) — double-precision to 32-bit	-
0	0	01	10	000	FCVTMS (scalar) — double-precision to 32-bit	-
0	0	01	10	001	FCVTMU (scalar) — double-precision to 32-bit	-
0	0	01	10	11x	UNALLOCATED	-
0	0	01	11	000	FCVTZS (scalar, integer) — double-precision to 32-bit	-
0	0	01	11	001	FCVTZU (scalar, integer) — double-precision to 32-bit	-
0	0	01	11	110	FJCVTZS	FEAT_JSCVT
0	0	01	11	111	UNALLOCATED	-
0	0	10		11x	UNALLOCATED	-
0	0	11	00	000	FCVTNS (scalar) — half-precision to 32-bit	FEAT_FP16
0	0	11	00	001	FCVTNU (scalar) — half-precision to 32-bit	FEAT_FP16
0	0	11	00	010	SCVTF (scalar, integer) — 32-bit to half-precision	FEAT_FP16
0	0	11	00	011	UCVTF (scalar, integer) — 32-bit to half-precision	FEAT_FP16
0	0	11	00	100	FCVTAS (scalar) — half-precision to 32-bit	FEAT_FP16
0	0	11	00	101	FCVTAU (scalar) — half-precision to 32-bit	FEAT_FP16
0	0	11	00	110	FMOV (general) — half-precision to 32-bit	FEAT_FP16
0	0	11	00	111	FMOV (general) — 32-bit to half-precision	FEAT_FP16
0	0	11	01	000	FCVTPS (scalar) — half-precision to 32-bit	FEAT_FP16
0	0	11	01	001	FCVTPU (scalar) — half-precision to 32-bit	FEAT_FP16
0	0	11	10	000	FCVTMS (scalar) — half-precision to 32-bit	FEAT_FP16
0	0	11	10	001	FCVTMU (scalar) — half-precision to 32-bit	FEAT_FP16
0	0	11	11	000	FCVTZS (scalar, integer) — half-precision to 32-bit	FEAT_FP16
0	0	11	11	001	FCVTZU (scalar, integer) — half-precision to 32-bit	FEAT_FP16
1	0	00		11x	UNALLOCATED	-
1	0	00	00	000	FCVTNS (scalar) — single-precision to 64-bit	-
1	0	00	00	001	FCVTNU (scalar) — single-precision to 64-bit	-
1	0	00	00	010	SCVTF (scalar, integer) — 64-bit to single-precision	-
1	0	00	00	011	UCVTF (scalar, integer) — 64-bit to single-precision	-
1	0	00	00	100	FCVTAS (scalar) — single-precision to 64-bit	-
1	0	00	00	101	FCVTAU (scalar) — single-precision to 64-bit	-
1	0	00	01	000	FCVTPS (scalar) — single-precision to 64-bit	-

Decode fields					Instruction Details	Feature
sf	S	ptype	rmode	opcode		
1	0	00	01	001	FCVTPU (scalar) — single-precision to 64-bit	-
1	0	00	10	000	FCVTMS (scalar) — single-precision to 64-bit	-
1	0	00	10	001	FCVTMU (scalar) — single-precision to 64-bit	-
1	0	00	11	000	FCVTZS (scalar, integer) — single-precision to 64-bit	-
1	0	00	11	001	FCVTZU (scalar, integer) — single-precision to 64-bit	-
1	0	01	x1	11x	UNALLOCATED	-
1	0	01	00	000	FCVTNS (scalar) — double-precision to 64-bit	-
1	0	01	00	001	FCVTNU (scalar) — double-precision to 64-bit	-
1	0	01	00	010	SCVTF (scalar, integer) — 64-bit to double-precision	-
1	0	01	00	011	UCVTF (scalar, integer) — 64-bit to double-precision	-
1	0	01	00	100	FCVTAS (scalar) — double-precision to 64-bit	-
1	0	01	00	101	FCVTAU (scalar) — double-precision to 64-bit	-
1	0	01	00	110	FMOV (general) — double-precision to 64-bit	-
1	0	01	00	111	FMOV (general) — 64-bit to double-precision	-
1	0	01	01	000	FCVTPS (scalar) — double-precision to 64-bit	-
1	0	01	01	001	FCVTPU (scalar) — double-precision to 64-bit	-
1	0	01	1x	11x	UNALLOCATED	-
1	0	01	10	000	FCVTMS (scalar) — double-precision to 64-bit	-
1	0	01	10	001	FCVTMU (scalar) — double-precision to 64-bit	-
1	0	01	11	000	FCVTZS (scalar, integer) — double-precision to 64-bit	-
1	0	01	11	001	FCVTZU (scalar, integer) — double-precision to 64-bit	-
1	0	10	x0	11x	UNALLOCATED	-
1	0	10	01	110	FMOV (general) — top half of 128-bit to 64-bit	-
1	0	10	01	111	FMOV (general) — 64-bit to top half of 128-bit	-
1	0	10	1x	11x	UNALLOCATED	-
1	0	11	00	000	FCVTNS (scalar) — half-precision to 64-bit	FEAT_FP16
1	0	11	00	001	FCVTNU (scalar) — half-precision to 64-bit	FEAT_FP16
1	0	11	00	010	SCVTF (scalar, integer) — 64-bit to half-precision	FEAT_FP16
1	0	11	00	011	UCVTF (scalar, integer) — 64-bit to half-precision	FEAT_FP16
1	0	11	00	100	FCVTAS (scalar) — half-precision to 64-bit	FEAT_FP16
1	0	11	00	101	FCVTAU (scalar) — half-precision to 64-bit	FEAT_FP16
1	0	11	00	110	FMOV (general) — half-precision to 64-bit	FEAT_FP16
1	0	11	00	111	FMOV (general) — 64-bit to half-precision	FEAT_FP16
1	0	11	01	000	FCVTPS (scalar) — half-precision to 64-bit	FEAT_FP16
1	0	11	01	001	FCVTPU (scalar) — half-precision to 64-bit	FEAT_FP16
1	0	11	10	000	FCVTMS (scalar) — half-precision to 64-bit	FEAT_FP16
1	0	11	10	001	FCVTMU (scalar) — half-precision to 64-bit	FEAT_FP16
1	0	11	11	000	FCVTZS (scalar, integer) — half-precision to 64-bit	FEAT_FP16
1	0	11	11	001	FCVTZU (scalar, integer) — half-precision to 64-bit	FEAT_FP16

Floating-point data-processing (1 source)

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	0	S	1	1	1	1	0	ptype		1	opcode						1	0	0	0	0	Rn					Rd					

Decode fields				Instruction Details	Feature
M	S	ptype	opcode		
			1xxxxx	UNALLOCATED	-
	1			UNALLOCATED	-
0	0	00	000000	FMOV (register) — single-precision	-
0	0	00	000001	FABS (scalar) — single-precision	-
0	0	00	000010	FNEG (scalar) — single-precision	-
0	0	00	000011	FSQRT (scalar) — single-precision	-
0	0	00	000100	UNALLOCATED	-
0	0	00	000101	FCVT — single-precision to double-precision	-
0	0	00	000110	UNALLOCATED	-
0	0	00	000111	FCVT — single-precision to half-precision	-
0	0	00	001000	FRINTN (scalar) — single-precision	-
0	0	00	001001	FRINTP (scalar) — single-precision	-
0	0	00	001010	FRINTM (scalar) — single-precision	-
0	0	00	001011	FRINTZ (scalar) — single-precision	-
0	0	00	001100	FRINTA (scalar) — single-precision	-
0	0	00	001101	UNALLOCATED	-
0	0	00	001110	FRINTX (scalar) — single-precision	-
0	0	00	001111	FRINTI (scalar) — single-precision	-
0	0	00	010000	FRINT32Z (scalar) — single-precision	FEAT_FRINTTS
0	0	00	010001	FRINT32X (scalar) — single-precision	FEAT_FRINTTS
0	0	00	010010	FRINT64Z (scalar) — single-precision	FEAT_FRINTTS
0	0	00	010011	FRINT64X (scalar) — single-precision	FEAT_FRINTTS
0	0	00	0101xx	UNALLOCATED	-
0	0	00	011xxx	UNALLOCATED	-
0	0	01	000000	FMOV (register) — double-precision	-
0	0	01	000001	FABS (scalar) — double-precision	-
0	0	01	000010	FNEG (scalar) — double-precision	-
0	0	01	000011	FSQRT (scalar) — double-precision	-
0	0	01	000100	FCVT — double-precision to single-precision	-
0	0	01	000101	UNALLOCATED	-
0	0	01	000110	BFCVT	FEAT_BF16
0	0	01	000111	FCVT — double-precision to half-precision	-
0	0	01	001000	FRINTN (scalar) — double-precision	-
0	0	01	001001	FRINTP (scalar) — double-precision	-
0	0	01	001010	FRINTM (scalar) — double-precision	-
0	0	01	001011	FRINTZ (scalar) — double-precision	-
0	0	01	001100	FRINTA (scalar) — double-precision	-
0	0	01	001101	UNALLOCATED	-
0	0	01	001110	FRINTX (scalar) — double-precision	-
0	0	01	001111	FRINTI (scalar) — double-precision	-
0	0	01	010000	FRINT32Z (scalar) — double-precision	FEAT_FRINTTS
0	0	01	010001	FRINT32X (scalar) — double-precision	FEAT_FRINTTS
0	0	01	010010	FRINT64Z (scalar) — double-precision	FEAT_FRINTTS
0	0	01	010011	FRINT64X (scalar) — double-precision	FEAT_FRINTTS
0	0	01	0101xx	UNALLOCATED	-
0	0	01	011xxx	UNALLOCATED	-
0	0	10	0xxxxx	UNALLOCATED	-

Decode fields				Instruction Details	Feature
M	S	ptype	opcode		
0	0	11	000000	FMOV (register) — half-precision	FEAT_FP16
0	0	11	000001	FABS (scalar) — half-precision	FEAT_FP16
0	0	11	000010	FNEG (scalar) — half-precision	FEAT_FP16
0	0	11	000011	FSQRT (scalar) — half-precision	FEAT_FP16
0	0	11	000100	FCVT — half-precision to single-precision	-
0	0	11	000101	FCVT — half-precision to double-precision	-
0	0	11	00011x	UNALLOCATED	-
0	0	11	001000	FRINTN (scalar) — half-precision	FEAT_FP16
0	0	11	001001	FRINTP (scalar) — half-precision	FEAT_FP16
0	0	11	001010	FRINTM (scalar) — half-precision	FEAT_FP16
0	0	11	001011	FRINTZ (scalar) — half-precision	FEAT_FP16
0	0	11	001100	FRINTA (scalar) — half-precision	FEAT_FP16
0	0	11	001101	UNALLOCATED	-
0	0	11	001110	FRINTX (scalar) — half-precision	FEAT_FP16
0	0	11	001111	FRINTI (scalar) — half-precision	FEAT_FP16
0	0	11	01xxxx	UNALLOCATED	-
1				UNALLOCATED	-

Floating-point compare

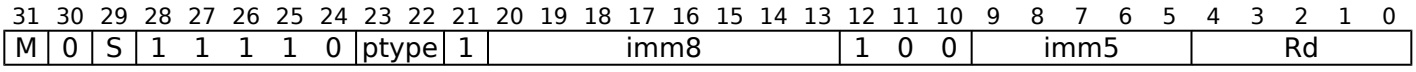
These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M	0	S	1	1	1	1	0	ptype		1	Rm					op		1	0	0	0	Rn					opcode2				

Decode fields					Instruction Details	Feature
M	S	ptype	op	opcode2		
				xxxx1	UNALLOCATED	-
				xxx1x	UNALLOCATED	-
				xx1xx	UNALLOCATED	-
			x1		UNALLOCATED	-
			1x		UNALLOCATED	-
		10			UNALLOCATED	-
	1				UNALLOCATED	-
0	0	00	00	00000	FCMP	-
0	0	00	00	01000	FCMP	-
0	0	00	00	10000	FCMPE	-
0	0	00	00	11000	FCMPE	-
0	0	01	00	00000	FCMP	-
0	0	01	00	01000	FCMP	-
0	0	01	00	10000	FCMPE	-
0	0	01	00	11000	FCMPE	-
0	0	11	00	00000	FCMP	FEAT_FP16
0	0	11	00	01000	FCMP	FEAT_FP16
0	0	11	00	10000	FCMPE	FEAT_FP16
0	0	11	00	11000	FCMPE	FEAT_FP16
1					UNALLOCATED	-

Floating-point immediate

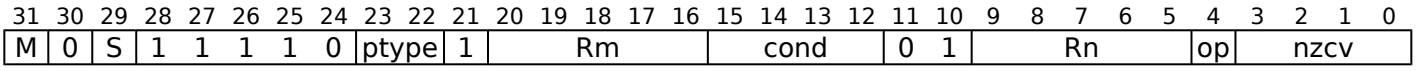
These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.



Decode fields				Instruction Details	Feature
M	S	ptype	imm5		
			xxxx1	UNALLOCATED	-
			xxx1x	UNALLOCATED	-
			xx1xx	UNALLOCATED	-
			x1xxx	UNALLOCATED	-
			1xxxx	UNALLOCATED	-
		10		UNALLOCATED	-
	1			UNALLOCATED	-
0	0	00	00000	FMOV (scalar, immediate) — single-precision	-
0	0	01	00000	FMOV (scalar, immediate) — double-precision	-
0	0	11	00000	FMOV (scalar, immediate) — half-precision	FEAT_FP16
1				UNALLOCATED	-

Floating-point conditional compare

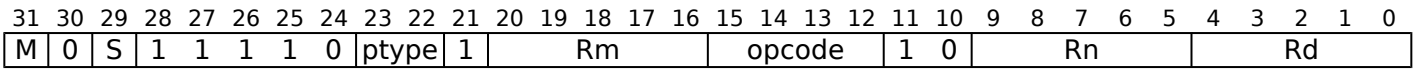
These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.



Decode fields				Instruction Details	Feature
M	S	ptype	op		
		10		UNALLOCATED	-
	1			UNALLOCATED	-
0	0	00	0	FCCMP — single-precision	-
0	0	00	1	FCCMPE — single-precision	-
0	0	01	0	FCCMP — double-precision	-
0	0	01	1	FCCMPE — double-precision	-
0	0	11	0	FCCMP — half-precision	FEAT_FP16
0	0	11	1	FCCMPE — half-precision	FEAT_FP16
1				UNALLOCATED	-

Floating-point data-processing (2 source)

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.

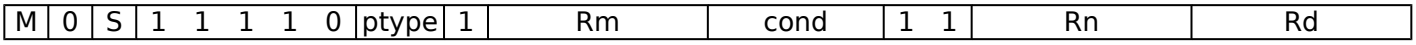


Decode fields				Instruction Details	Feature
M	S	ptype	opcode		
			1xx1	UNALLOCATED	-
			1x1x	UNALLOCATED	-
			11xx	UNALLOCATED	-
		10		UNALLOCATED	-
	1			UNALLOCATED	-
0	0	00	0000	FMUL (scalar) — single-precision	-

Decode fields				Instruction Details	Feature
M	S	ptype	opcode		
0	0	00	0001	FDIV (scalar) — single-precision	-
0	0	00	0010	FADD (scalar) — single-precision	-
0	0	00	0011	FSUB (scalar) — single-precision	-
0	0	00	0100	FMAX (scalar) — single-precision	-
0	0	00	0101	FMIN (scalar) — single-precision	-
0	0	00	0110	FMAXNM (scalar) — single-precision	-
0	0	00	0111	FMINNM (scalar) — single-precision	-
0	0	00	1000	FNMUL (scalar) — single-precision	-
0	0	01	0000	FMUL (scalar) — double-precision	-
0	0	01	0001	FDIV (scalar) — double-precision	-
0	0	01	0010	FADD (scalar) — double-precision	-
0	0	01	0011	FSUB (scalar) — double-precision	-
0	0	01	0100	FMAX (scalar) — double-precision	-
0	0	01	0101	FMIN (scalar) — double-precision	-
0	0	01	0110	FMAXNM (scalar) — double-precision	-
0	0	01	0111	FMINNM (scalar) — double-precision	-
0	0	01	1000	FNMUL (scalar) — double-precision	-
0	0	11	0000	FMUL (scalar) — half-precision	FEAT_FP16
0	0	11	0001	FDIV (scalar) — half-precision	FEAT_FP16
0	0	11	0010	FADD (scalar) — half-precision	FEAT_FP16
0	0	11	0011	FSUB (scalar) — half-precision	FEAT_FP16
0	0	11	0100	FMAX (scalar) — half-precision	FEAT_FP16
0	0	11	0101	FMIN (scalar) — half-precision	FEAT_FP16
0	0	11	0110	FMAXNM (scalar) — half-precision	FEAT_FP16
0	0	11	0111	FMINNM (scalar) — half-precision	FEAT_FP16
0	0	11	1000	FNMUL (scalar) — half-precision	FEAT_FP16
1				UNALLOCATED	-

Floating-point conditional select

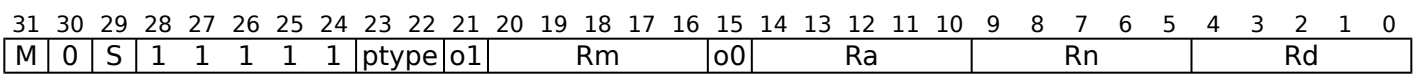
These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.



Decode fields			Instruction Details	Feature
M	S	ptype		
		10	UNALLOCATED	-
	1		UNALLOCATED	-
0	0	00	FCSEL — single-precision	-
0	0	01	FCSEL — double-precision	-
0	0	11	FCSEL — half-precision	FEAT_FP16
1			UNALLOCATED	-

Floating-point data-processing (3 source)

These instructions are under Data Processing -- Scalar Floating-Point and Advanced SIMD.



Decode fields					Instruction Details	Feature
M	S	ptype	o1	o0		
		10			UNALLOCATED	-
	1				UNALLOCATED	-
0	0	00	0	0	FMADD — single-precision	-
0	0	00	0	1	FMSUB — single-precision	-
0	0	00	1	0	FNMADD — single-precision	-
0	0	00	1	1	FNMSUB — single-precision	-
0	0	01	0	0	FMADD — double-precision	-
0	0	01	0	1	FMSUB — double-precision	-
0	0	01	1	0	FNMADD — double-precision	-
0	0	01	1	1	FNMSUB — double-precision	-
0	0	11	0	0	FMADD — half-precision	FEAT_FP16
0	0	11	0	1	FMSUB — half-precision	FEAT_FP16
0	0	11	1	0	FNMADD — half-precision	FEAT_FP16
0	0	11	1	1	FNMSUB — half-precision	FEAT_FP16
1					UNALLOCATED	-

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51

Shared Pseudocode Functions

This page displays common pseudocode functions shared by many pages

Pseudocodes

Library pseudocode for aarch32/at/AArch32.AT

```
// AArch32.AT()
// ============
// Perform address translation as per AT instructions.

AArch32.AT(bits(32) vaddress, TranslationStage stage_in, bits(2) el, ATAccess ataccess)
    TranslationStage stage = stage_in;
    SecurityState ss;
    Regime regime;
    boolean eae;

    // ATS1Hx instructions
    if el == EL2 then
        regime = Regime_EL2;
        eae = TRUE;
        ss = SS_NonSecure;

    // ATS1Cxx instructions
    elsif stage == TranslationStage_1 || (stage == TranslationStage_12 && !HaveEL(EL2)) then
        stage = TranslationStage_1;
        ss = SecurityStateAtEL(PSTATE.EL);
        regime = if ss == SS_Secure && ELUsingAArch32(EL3) then Regime_EL30 else Regime_EL10;
        eae = TTBCR.EAE == '1';

    // ATS12NSOxx instructions
    else
        regime = Regime_EL10;
        eae = if HaveAArch32EL(EL3) then TTBCR_NS.EAE == '1' else TTBCR.EAE == '1';
        ss = SS_NonSecure;

    AddressDescriptor addrdesc;
    SDFType sdftype;
    aligned = TRUE;
    ispriv = el != EL0;
    supersection = '0';
    iswrite = ataccess IN {ATAccess_WritePAN, ATAccess_Write};
    acctype = if ataccess IN {ATAccess_Read, ATAccess_Write} then AccType_AT else AccType_ATPAN;

    // Prepare fault fields in case a fault is detected
    fault = NoFault();
    fault.acctype = acctype;
    fault.write = iswrite;

    if eae then
        (fault, addrdesc) = AArch32.S1TranslateLD(fault, regime, ss, vaddress, acctype, aligned,
                                                  iswrite, ispriv);
    else
        (fault, addrdesc, sdftype) = AArch32.S1TranslateSD(fault, regime, ss, vaddress, acctype,
                                                           aligned, iswrite, ispriv);
        supersection = if sdftype == SDFType_Supersection then '1' else '0';

    // ATS12NSOxx instructions
    if stage == TranslationStage_12 && fault.statuscode == Fault_None then
        s2fs1walk = FALSE;
        (fault, addrdesc) = AArch32.S2Translate(fault, addrdesc, ss, s2fs1walk, acctype, aligned,
                                                iswrite, ispriv);

    if fault.statuscode != Fault_None then
        // Take exception when External abort occurs on translation table walk
        if (IsExternalAbort(fault) || (stage == TranslationStage_1 && el != EL2 && PSTATE.EL == EL1
                                       && EL2Enabled() && fault.s2fs1walk)) then
            PAR = bits(64) UNKNOWN;
            AArch32.Abort(vaddress, fault);

    addrdesc.fault = fault;

    if (eae || (stage == TranslationStage_12 && (HCR.VM == '1' || HCR.DC == '1'))
            || (stage == TranslationStage_1 && el != EL2 && PSTATE.EL == EL2)) then
        AArch32.EncodePARLD(addrdesc, ss);
    else
```

```
        AArch32.EncodePARSD(addrdesc, supersection, ss);
    return;
```

Library pseudocode for aarch32/at/AArch32.EncodePARLD

```
// AArch32.EncodePARLD()
// =====================
// Returns 64-bit format PAR on address translation instruction.

AArch32.EncodePARLD(AddressDescriptor addrdesc, SecurityState ss)

    if !IsFault(addrdesc) then
        bit ns;
        if ss == SS_NonSecure then
            ns = bit UNKNOWN;
        elsif addrdesc.paddress.paspace == PAS_Secure then
            ns = '0';
        else
            ns = '1';
        PAR.F      = '0';
        PAR.SH     = ReportedPARShareability(PAREncodeShareability(addrdesc.memattrs));
        PAR.NS     = ns;
        PAR<10>    = bit IMPLEMENTATION_DEFINED "Non-Faulting PAR";               // IMPDEF
        PAR.LPAE   = '1';
        PAR.PA     = addrdesc.paddress.address<39:12>;
        PAR.ATTR   = ReportedPARAttrs(EncodePARAttrs(addrdesc.memattrs));
    else
        PAR.F      = '1';
        PAR.FST    = AArch32.PARFaultStatusLD(addrdesc.fault);
        PAR.S2WLK  = if addrdesc.fault.s2fs1walk then '1' else '0';
        PAR.FSTAGE = if addrdesc.fault.secondstage then '1' else '0';
        PAR.LPAE   = '1';
        PAR<63:48> = bits(16) IMPLEMENTATION_DEFINED "Faulting PAR";              // IMPDEF
    return;
```

Library pseudocode for aarch32/at/AArch32.EncodePARSD

```
// AArch32.EncodePARSD()
// =====================
// Returns 32-bit format PAR on address translation instruction.

AArch32.EncodePARSD(AddressDescriptor addrdesc_in, bit supersection, SecurityState ss)
    AddressDescriptor addrdesc = addrdesc_in;
    if !IsFault(addrdesc) then
        if (addrdesc.mematttrs.memtype == MemType_Device ||
                (addrdesc.mematttrs.inner.attrs == MemAttr_NC &&
                 addrdesc.mematttrs.outer.attrs == MemAttr_NC)) then
            addrdesc.mematttrs.shareability = Shareability_OSH;
        bit ns;
        if ss == SS_NonSecure then
            ns = bit UNKNOWN;
        elsif addrdesc.paddress.paspace == PAS_Secure then
            ns = '0';
        else
            ns = '1';
        bits(2) sh = if addrdesc.mematttrs.shareability != Shareability_NSH then '01' else '00';
        PAR.F      = '0';
        PAR.SS     = supersection;
        PAR.Outer  = AArch32.ReportedOuterAttrs(AArch32.PAROuterAttrs(addrdesc.mematttrs));
        PAR.Inner  = AArch32.ReportedInnerAttrs(AArch32.PARInnerAttrs(addrdesc.mematttrs));
        PAR.SH     = ReportedPARShareability(sh);
        PAR<8>     = bit IMPLEMENTATION_DEFINED "Non-Faulting PAR";               // IMPDEF
        PAR.NS     = ns;
        PAR.NOS    = if addrdesc.mematttrs.shareability == Shareability_OSH then '0' else '1';
        PAR.LPAE   = '0';
        PAR.PA     = addrdesc.paddress.address<39:12>;
    else
        PAR.F      = '1';
        PAR.FST    = AArch32.PARFaultStatusSD(addrdesc.fault);
        PAR.LPAE   = '0';
        PAR<31:16> = bits(16) IMPLEMENTATION_DEFINED "Faulting PAR";              // IMPDEF
    return;
```

Library pseudocode for aarch32/at/AArch32.PARFaultStatusLD

```
// AArch32.PARFaultStatusLD()
// ==========================
// Fault status field decoding of 64-bit PAR

bits(6) AArch32.PARFaultStatusLD(FaultRecord fault)
    bits(32) syndrome;

    if fault.statuscode == Fault_Domain then
        // Report Domain fault
        assert fault.level IN {1,2};
        syndrome<1:0> = if fault.level == 1 then '01' else '10';
        syndrome<5:2> = '1111';
    else
        syndrome = AArch32.FaultStatusLD(TRUE, fault);
    return syndrome<5:0>;
```

Library pseudocode for aarch32/at/AArch32.PARFaultStatusSD

```
// AArch32.PARFaultStatusSD()
// ==========================
// Fault status field decoding of 32-bit PAR.

bits(6) AArch32.PARFaultStatusSD(FaultRecord fault)
    bits(32) syndrome;

    syndrome = AArch32.FaultStatusSD(TRUE, fault);
    return syndrome<12,10,3:0>;
```

Library pseudocode for aarch32/at/AArch32.PARInnerAttrs

```
// AArch32.PARInnerAttrs()
// ======================
// Convert orthogonal attributes and hints to 32-bit PAR Inner field.

bits(3) AArch32.PARInnerAttrs(MemoryAttributes memattrs)
    bits(3) result;

    if memattrs.memtype == MemType_Device then
        if memattrs.device == DeviceType_nGnRnE then
            result = '001'; // Non-cacheable
        elsif memattrs.device == DeviceType_nGnRE then
            result = '011'; // Non-cacheable
    else
        MemAttrHints inner = memattrs.inner;
        if inner.attrs == MemAttr_NC then
            result = '000'; // Non-cacheable
        elsif inner.attrs == MemAttr_WB && inner.hints<0> == '1' then
            result = '101'; // Write-Back, Write-Allocate
        elsif inner.attrs == MemAttr_WT then
            result = '110'; // Write-Through
        elsif inner.attrs == MemAttr_WB && inner.hints<0> == '0' then
            result = '111'; // Write-Back, no Write-Allocate
    return result;
```

Library pseudocode for aarch32/at/AArch32.PAROuterAttrs

```
// AArch32.PAROuterAttrs()
// ======================
// Convert orthogonal attributes and hints to 32-bit PAR Outer field.

bits(2) AArch32.PAROuterAttrs(MemoryAttributes memattrs)
    bits(2) result;

    if memattrs.memtype == MemType_Device then
        result = bits(2) UNKNOWN;
    else
        MemAttrHints outer = memattrs.outer;
        if outer.attrs == MemAttr_NC then
            result = '00'; // Non-cacheable
        elsif outer.attrs == MemAttr_WB && outer.hints<0> == '1' then
            result = '01'; // Write-Back, Write-Allocate
        elsif outer.attrs == MemAttr_WT && outer.hints<0> == '0' then
            result = '10'; // Write-Through, no Write-Allocate
        elsif outer.attrs == MemAttr_WB && outer.hints<0> == '0' then
            result = '11'; // Write-Back, no Write-Allocate
    return result;
```

Library pseudocode for aarch32/dc/AArch32.DC

```
// AArch32.DC()
// ============
// Perform Data Cache Operation.

AArch32.DC(bits(32) regval, CacheOp cacheop, CacheOpScope opscope)
    AccType acctype = AccType_DC;
    CacheRecord cache;

    cache.acctype = acctype;
    cache.cacheop = cacheop;
    cache.opscope = opscope;
    cache.cachetype = CacheType_Data;
    cache.security = SecurityStateAtEL(PSTATE.EL);

    if opscope == CacheOpScope_SetWay then
        cache.shareability = Shareability_NSH;
        (cache.set, cache.way, cache.level) = DecodeSW(ZeroExtend(regval, 64), CacheType_Data);

        if (cacheop == CacheOp_Invalidate && PSTATE.EL == EL1 && EL2Enabled() &&
          ((!ELUsingAArch32(EL2) && HCR_EL2.SWIO == '1') || (ELUsingAArch32(EL2) && HCR.SWIO == '1') ||
          (!ELUsingAArch32(EL2) && HCR_EL2.<DC,VM> != '00') || (ELUsingAArch32(EL2) && HCR.<DC,VM> != '00
            cache.cacheop = CacheOp_CleanInvalidate;
        CACHE_OP(cache);
        return;

    if EL2Enabled() then
        if PSTATE.EL IN {EL0, EL1} then
            cache.is_vmid_valid = TRUE;
            cache.vmid          = VMID[];
        else
            cache.is_vmid_valid = FALSE;
    else
        cache.is_vmid_valid = FALSE;

    if PSTATE.EL == EL0 then
        cache.is_asid_valid = TRUE;
        cache.asid          = ASID[];
    else
        cache.is_asid_valid = FALSE;

    need_translate = DCInstNeedsTranslation(opscope);
    iswrite = cacheop == CacheOp_Invalidate;
    vaddress = regval;

    size = 0;        // by default no watchpoint address
    if iswrite then
        size = integer IMPLEMENTATION_DEFINED "Data Cache Invalidate Watchpoint Size";
        assert size >= 4*(2^(UInt(CTR_EL0.DminLine))) && size <= 2048;
        assert UInt(size<32:0> AND (size-1)<32:0>) == 0; // size is power of 2
        vaddress = Align(regval, size);

    cache.translated = need_translate;
    cache.vaddress = ZeroExtend(vaddress, 64);

    if need_translate then
        wasaligned = TRUE;
        memaddrdesc = AArch32.TranslateAddress(vaddress, acctype, iswrite, wasaligned, size);
        if IsFault(memaddrdesc) then
            AArch32.Abort(regval, memaddrdesc.fault);

        memattrs = memaddrdesc.memattrs;
        cache.paddress = memaddrdesc.paddress;
        if opscope == CacheOpScope_PoC then
            cache.shareability = memattrs.shareability;
        else
            cache.shareability = Shareability_NSH;
    else
        cache.shareability = Shareability UNKNOWN;
        cache.paddress = FullAddress UNKNOWN;
```

```
    if (cacheop == CacheOp_Invalidate && PSTATE.EL == EL1 && EL2Enabled()
      && ((!ELUsingAArch32(EL2) && HCR_EL2.<DC,VM> != '00') || (ELUsingAArch32(EL2) && HCR.<DC,VM> != '00
        cache.cacheop = CacheOp_CleanInvalidate;

    CACHE_OP(cache);
    return;
```

Library pseudocode for aarch32/debug/VCRMatch/AArch32.VCRMatch

```
// AArch32.VCRMatch()
// ==================

boolean AArch32.VCRMatch(bits(32) vaddress)

    boolean match;
    if UsingAArch32() && ELUsingAArch32(EL1) && PSTATE.EL != EL2 then
        // Each bit position in this string corresponds to a bit in DBGVCR and an exception vector.
        match_word = Zeros(32);

        ss = CurrentSecurityState();
        if vaddress<31:5> == ExcVectorBase()<31:5> then
            if HaveEL(EL3) && ss == SS_NonSecure then
                match_word<UInt(vaddress<4:2>) + 24> = '1';     // Non-secure vectors
            else
                match_word<UInt(vaddress<4:2>) + 0> = '1';      // Secure vectors (or no EL3)

        if (HaveEL(EL3) && ELUsingAArch32(EL3) && vaddress<31:5> == MVBAR<31:5> &&
            ss == SS_Secure) then
            match_word<UInt(vaddress<4:2>) + 8> = '1';          // Monitor vectors

        // Mask out bits not corresponding to vectors.
        bits(32) mask;
        if !HaveEL(EL3) then
            mask = '00000000':'00000000':'00000000':'11011110'; // DBGVCR[31:8] are RES0
        elsif !ELUsingAArch32(EL3) then
            mask = '11011110':'00000000':'00000000':'11011110'; // DBGVCR[15:8] are RES0
        else
            mask = '11011110':'00000000':'11011100':'11011110';

        match_word = match_word AND DBGVCR AND mask;
        match = !IsZero(match_word);

        // Check for UNPREDICTABLE case - match on Prefetch Abort and Data Abort vectors
        if !IsZero(match_word<28:27,12:11,4:3>) && DebugTarget() == PSTATE.EL then
            match = ConstrainUnpredictableBool(Unpredictable_VCMATCHDAPA);

        if !IsZero(vaddress<1:0>) && match then
            match = ConstrainUnpredictableBool(Unpredictable_VCMATCHHALF);
    else
        match = FALSE;

    return match;
```

Library pseudocode for aarch32/debug/authentication/ AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled

```
// AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
// ========================================================

boolean AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
    // The definition of this function is IMPLEMENTATION DEFINED.
    // In the recommended interface, AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled returns
    // the state of the (DBGEN AND SPIDEN) signal.
    if !HaveEL(EL3) && NonSecureOnlyImplementation() then return FALSE;
    return DBGEN == HIGH && SPIDEN == HIGH;
```

Library pseudocode for aarch32/debug/breakpoint/AArch32.BreakpointMatch

```
// AArch32.BreakpointMatch()
// =========================
// Breakpoint matching in an AArch32 translation regime.

(boolean,boolean) AArch32.BreakpointMatch(integer n, bits(32) vaddress,
                                          integer size)
    assert ELUsingAArch32(S1TranslationRegime());
    assert n < NumBreakpointsImplemented();

    enabled = DBGBCR[n].E == '1';
    ispriv = PSTATE.EL != EL0;
    linked = DBGBCR[n].BT IN {'0x01'};
    isbreakpnt = TRUE;
    linked_to = FALSE;

    state_match = AArch32.StateMatch(DBGBCR[n].SSC, DBGBCR[n].HMC, DBGBCR[n].PMC,
                                     linked, DBGBCR[n].LBN, isbreakpnt,  ispriv);
    (value_match, value_mismatch) = AArch32.BreakpointValueMatch(n, vaddress, linked_to);

    if size == 4 then                    // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
        // event.
        (match_i, mismatch_i) = AArch32.BreakpointValueMatch(n, vaddress + 2, linked_to);
        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);
        if value_mismatch && !mismatch_i then
            value_mismatch = ConstrainUnpredictableBool(Unpredictable_BPMISMATCHHALF);
    if vaddress<1> == '1' && DBGBCR[n].BAS == '1111' then
        // The above notwithstanding, if DBGBCR[n].BAS == '1111', then it is CONSTRAINED
        // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
        // at the address DBGBVR[n]+2.
        if value_match then value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);
        if !value_mismatch then value_mismatch = ConstrainUnpredictableBool(Unpredictable_BPMISMATCHHALF)

    match = value_match && state_match && enabled;
    mismatch = value_mismatch && state_match && enabled;

    return (match, mismatch);
```

Library pseudocode for aarch32/debug/breakpoint/AArch32.BreakpointValueMatch

```
// AArch32.BreakpointValueMatch()
// ==============================
// The first result is whether an Address Match or Context breakpoint is programmed on the
// instruction at "address". The second result is whether an Address Mismatch breakpoint is
// programmed on the instruction, that is, whether the instruction should be stepped.

(boolean,boolean) AArch32.BreakpointValueMatch(integer n_in, bits(32) vaddress, boolean linked_to)

    // "n" is the identity of the breakpoint unit to match against.
    // "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
    //   matching breakpoints.
    // "linked_to" is TRUE if this is a call from StateMatch for linking.
    integer n = n_in;

    // If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
    // no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
    if n >= NumBreakpointsImplemented() then
        Constraint c;
        (c, n) = ConstrainUnpredictableInteger(0, NumBreakpointsImplemented() - 1, Unpredictable_BPNOTIMP
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return (FALSE,FALSE);

    // If this breakpoint is not enabled, it cannot generate a match. (This could also happen on a
    // call from StateMatch for linking).
    if DBGBCR[n].E == '0' then return (FALSE,FALSE);

    context_aware = (n >= (NumBreakpointsImplemented() - NumContextAwareBreakpointsImplemented()));

    // If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
    dbgtype = DBGBCR[n].BT;
    Constraint c;

    if ((dbgtype IN {'011x','11xx'} && !HaveVirtHostExt() && !HaveV82Debug()) ||    // Context matching
          (dbgtype IN {'010x'} && HaltOnBreakpointOrWatchpoint()) ||                  // Address mismatch
          (!(dbgtype IN {'0x0x'}) && !context_aware) ||                                  // Context match
          (dbgtype IN {'1xxx'} && !HaveEL(EL2))) then                                 // EL2 extension
        (c, dbgtype) = ConstrainUnpredictableBits(Unpredictable_RESBPTYPE, 4);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return (FALSE,FALSE);
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    // Determine what to compare against.
    match_addr = (dbgtype IN {'0x0x'});
    mismatch   = (dbgtype IN {'010x'});
    match_vmid = (dbgtype IN {'10xx'});
    match_cid1 = (dbgtype IN {'xx1x'});
    match_cid2 = (dbgtype IN {'11xx'});
    linked     = (dbgtype IN {'xxx1'});

    // If this is a call from StateMatch, return FALSE if the breakpoint is not programmed for a
    // VMID and/or context ID match, of if not context-aware. The above assertions mean that the
    // code can just test for match_addr == TRUE to confirm all these things.
    if linked_to && (!linked || match_addr) then return (FALSE,FALSE);

    // If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
    if !linked_to && linked && !match_addr then return (FALSE,FALSE);

    // Do the comparison.
    boolean BVR_match;
    if match_addr then
        boolean byte_select_match;
        byte = UInt(vaddress<1:0>);
        assert byte IN {0,2};                     // "vaddress" is halfword aligned
        byte_select_match = (DBGBCR[n].BAS<byte> == '1');
        integer top = 31;
        BVR_match = (vaddress<top:2> == DBGBVR[n]<top:2>) && byte_select_match;

    elsif match_cid1 then
        BVR_match = (PSTATE.EL != EL2 && CONTEXTIDR == DBGBVR[n]<31:0>);
    boolean BXVR_match;
```

```
    if match_vmid then
        bits(16) vmid;
        bits(16) bvr_vmid;
        if ELUsingAArch32(EL2) then
            vmid = ZeroExtend(VTTBR.VMID, 16);
            bvr_vmid = ZeroExtend(DBGBXVR[n]<7:0>, 16);
        elsif !Have16bitVMID() || VTCR_EL2.VS == '0' then
            vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
            bvr_vmid = ZeroExtend(DBGBXVR[n]<7:0>, 16);
        else
            vmid = VTTBR_EL2.VMID;
            bvr_vmid = DBGBXVR[n]<15:0>;
        BXVR_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                      vmid == bvr_vmid);
    elsif match_cid2 then
        BXVR_match = (PSTATE.EL != EL3 && (HaveVirtHostExt() || HaveV82Debug()) &&
                      EL2Enabled() &&
                      !ELUsingAArch32(EL2) &&
                      DBGBXVR[n]<31:0> == CONTEXTIDR_EL2<31:0>);

    bvr_match_valid = (match_addr || match_cid1);
    bxvr_match_valid = (match_vmid || match_cid2);

    match = (!bxvr_match_valid || BXVR_match) && (!bvr_match_valid || BVR_match);

    return (match && !mismatch, !match && mismatch);
```

Library pseudocode for aarch32/debug/breakpoint/AArch32.StateMatch

```
// AArch32.StateMatch()
// ====================
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch32.StateMatch(bits(2) SSC_in,  bit HMC_in,
                           bits(2) PxC_in, boolean linked_in, bits(4) LBN,
                           boolean isbreakpnt,  boolean ispriv)

    // "SSC_in","HMC_in","PxC_in" are the control fields from the DBGBCR[n] or DBGWCR[n] register.
    // "linked_in" is TRUE if this is a linked breakpoint/watchpoint type.
    // "LBN" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
    // "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.
    // "ispriv" is valid for watchpoints, and selects between privileged and unprivileged accesses.
    bits(2) SSC = SSC_in;
    bit HMC = HMC_in;
    bits(2) PxC = PxC_in;
    boolean linked = linked_in;

    // If parameters are set to a reserved type, behaves as either disabled or a defined type
    Constraint c;
    // SSCE value discarded as there is no SSCE bit in AArch32.
    (c, SSC, -, HMC, PxC) = CheckValidStateMatch(SSC, '0', HMC, PxC, isbreakpnt);
    if c == Constraint_DISABLED then return FALSE;
    // Otherwise the HMC,SSC,PxC values are either valid or the values returned by
    // CheckValidStateMatch are valid.

    PL2_match = HaveEL(EL2) && ((HMC == '1' && (SSC:PxC != '1000')) || SSC == '11');
    PL1_match = PxC<0> == '1';
    PL0_match = PxC<1> == '1';
    SSU_match = isbreakpnt && HMC == '0' && PxC == '00' && SSC != '11';

    boolean priv_match;
    if !ispriv && !isbreakpnt then
        priv_match = PL0_match;
    elsif SSU_match then
        priv_match = PSTATE.M IN {M32_User,M32_Svc,M32_System};
    else
        case PSTATE.EL of
            when EL3  priv_match = PL1_match;           // EL3 and EL1 are both PL1
            when EL2  priv_match = PL2_match;
            when EL1  priv_match = PL1_match;
            when EL0  priv_match = PL0_match;

    boolean security_state_match;
    ss = CurrentSecurityState();
    case SSC of
        when '00' security_state_match = TRUE;                          // Both
        when '01' security_state_match = ss == SS_NonSecure;            // Non-secure only
        when '10' security_state_match = ss == SS_Secure;               // Secure only
        when '11' security_state_match = (HMC == '1' || ss == SS_Secure);  // HMC=1 -> Both, 0 -> Secure
    integer lbn;
    if linked then
        // "LBN" must be an enabled context-aware breakpoint unit. If it is not context-aware then
        // it is CONSTRAINED UNPREDICTABLE whether this gives no match, or LBN is mapped to some
        // UNKNOWN breakpoint that is context-aware.
        lbn = UInt(LBN);
        first_ctx_cmp = NumBreakpointsImplemented() - NumContextAwareBreakpointsImplemented();
        last_ctx_cmp = NumBreakpointsImplemented() - 1;
        if (lbn < first_ctx_cmp || lbn > last_ctx_cmp) then
            (c, lbn) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp, Unpredictable_BPNOTCTXC
            assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
            case c of
                when Constraint_DISABLED  return FALSE;      // Disabled
                when Constraint_NONE      linked = FALSE;    // No linking
                // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint

    boolean linked_match;
    if linked then
        vaddress = bits(32) UNKNOWN;
        linked_to = TRUE;
```

```
        (linked_match,-) = AArch32.BreakpointValueMatch(lbn, vaddress, linked_to);

    return priv_match && security_state_match && (!linked || linked_match);
```

Library pseudocode for aarch32/debug/enables/AArch32.GenerateDebugExceptions

```
// AArch32.GenerateDebugExceptions()
// =================================

boolean AArch32.GenerateDebugExceptions()
    ss = CurrentSecurityState();
    return AArch32.GenerateDebugExceptionsFrom(PSTATE.EL, ss);
```

Library pseudocode for aarch32/debug/enables/AArch32.GenerateDebugExceptionsFrom

```
// AArch32.GenerateDebugExceptionsFrom()
// =====================================

boolean AArch32.GenerateDebugExceptionsFrom(bits(2) from_el, SecurityState from_state)

    if !ELUsingAArch32(DebugTargetFrom(from_state)) then
        mask = '0';    // No PSTATE.D in AArch32 state
        return AArch64.GenerateDebugExceptionsFrom(from_el, from_state, mask);

    if DBGOSLSR.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;

    boolean enabled;
    if HaveEL(EL3) && from_state == SS_Secure then
        assert from_el != EL2;    // Secure EL2 always uses AArch64
        if IsSecureEL2Enabled() then
            // Implies that EL3 and EL2 both using AArch64
            enabled = MDCR_EL3.SDD == '0';
        else
            spd = if ELUsingAArch32(EL3) then SDCR.SPD else MDCR_EL3.SPD32;
            if spd<1> == '1' then
                enabled = spd<0> == '1';
            else
                // SPD == 0b01 is reserved, but behaves the same as 0b00.
                enabled = AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled();
        if from_el == EL0 then enabled = enabled || SDER.SUIDEN == '1';
    else
        enabled = from_el != EL2;

    return enabled;
```

Library pseudocode for aarch32/debug/pmu/AArch32.CheckForPMUOverflow

```
// AArch32.CheckForPMUOverflow()
// =============================
// Signal Performance Monitors overflow IRQ and CTI overflow events

AArch32.CheckForPMUOverflow()
    if !ELUsingAArch32(EL1) then
        AArch64.CheckForPMUOverflow();
        return;
    bit hpme;
    if HaveEL(EL2) then
        hpme = if !ELUsingAArch32(EL2) then MDCR_EL2.HPME else HDCR.HPME;
    boolean pmuirq;
    bit E;
    pmuirq = PMCR.E == '1' && PMINTENSET.C == '1' && PMOVSSET.C == '1';
    integer counters = GetNumEventCounters();
    if counters != 0 then
        for idx = 0 to counters - 1
            E = if AArch32.PMUCounterIsHyp(idx) then hpme else PMCR.E;
            if E == '1' && PMINTENSET<idx> == '1' && PMOVSSET<idx> == '1' then pmuirq = TRUE;

    SetInterruptRequestLevel(InterruptID_PMUIRQ, if pmuirq then HIGH else LOW);

    CTI_SetEventLevel(CrossTriggerIn_PMUOverflow, if pmuirq then HIGH else LOW);

    // The request remains set until the condition is cleared. (For example, an interrupt handler
    // or cross-triggered event handler clears the overflow status flag by writing to PMOVSCLR.)
```

Library pseudocode for aarch32/debug/pmu/AArch32.ClearEventCounters

```
// AArch32.ClearEventCounters()
// ============================
// Zero all the event counters.

AArch32.ClearEventCounters()
    if HaveAArch64() then
        // Force the counter to be cleared as a 64-bit counter.
        AArch64.ClearEventCounters();
        return;

    integer counters = AArch32.GetNumEventCountersAccessible();
    if counters != 0 then
        for idx = 0 to counters - 1
            PMEVCNTR[idx] = Zeros(32);
```

Library pseudocode for aarch32/debug/pmu/AArch32.CountPMUEvents

```
// AArch32.CountPMUEvents()
// ========================
// Return TRUE if counter "idx" should count its event.
// For the cycle counter, idx == CYCLE_COUNTER_ID.

boolean AArch32.CountPMUEvents(integer idx)
    constant integer num_counters = GetNumEventCounters();
    assert idx == CYCLE_COUNTER_ID || idx < num_counters;
    if !ELUsingAArch32(EL1) then return AArch64.CountPMUEvents(idx);
    boolean debug;
    boolean enabled;
    boolean prohibited;
    boolean filtered;
    boolean frozen;
    boolean resvd_for_el2;
    bit E;
    bit spme;
    bits(32) ovflws;
    // Event counting is disabled in Debug state
    debug = Halted();

    // Software can reserve some counters for EL2
    resvd_for_el2 = AArch32.PMUCounterIsHyp(idx);
    ss = CurrentSecurityState();

    // Main enable controls
    if idx == CYCLE_COUNTER_ID then
        enabled = PMCR.E == '1' && PMCNTENSET.C == '1';
    else
        if resvd_for_el2 then
            E = if ELUsingAArch32(EL2) then HDCR.HPME else MDCR_EL2.HPME;
        else
            E = PMCR.E;
        enabled = E == '1' && PMCNTENSET<idx> == '1';

    // Event counting is allowed unless it is prohibited by any rule below
    prohibited = FALSE;

    // Event counting in Secure state is prohibited if all of:
    // * EL3 is implemented
    // * One of the following is true:
    //   - EL3 is using AArch64, MDCR_EL3.SPME == 0, and either:
    //     - FEAT_PMUv3p7 is not implemented
    //     - MDCR_EL3.MPMX == 0
    //   - EL3 is using AArch32 and SDCR.SPME == 0
    // * Not executing at EL0, or SDER.SUNIDEN == 0
    if HaveEL(EL3) && ss == SS_Secure then
        spme = if ELUsingAArch32(EL3) then SDCR.SPME else MDCR_EL3.SPME;
        if !ELUsingAArch32(EL3) && HavePMUv3p7() then
            prohibited = spme == '0' && MDCR_EL3.MPMX == '0';
        else
            prohibited = spme == '0';
        if prohibited && PSTATE.EL == EL0 then
            prohibited = SDER.SUNIDEN == '0';

    // Event counting at EL2 is prohibited if all of:
    // * The HPMD Extension is implemented
    // * PMNx is not reserved for EL2
    // * HDCR.HPMD == 1
    if !prohibited && PSTATE.EL == EL2 && HaveHPMDExt() && !resvd_for_el2 then
        prohibited = HDCR.HPMD == '1';

    // The IMPLEMENTATION DEFINED authentication interface might override software
    if prohibited && !HaveNoSecurePMUDisableOverride() then
        prohibited = !ExternalSecureNoninvasiveDebugEnabled();

    // Event counting might be frozen
    frozen = FALSE;

    // If FEAT_PMUv3p7 is implemented, event counting can be frozen
```

```
    if HavePMUv3p7() then
        bits(5) hpmn_bits;
        if HaveEL(EL2) then
            hpmn_bits = if !ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN;
        hpmn = UInt(hpmn_bits);
        ovflws = ZeroExtend(PMOVSSET<num_counters-1:0>, 32);
        bit FZ;
        if resvd_for_el2 then
            FZ = if ELUsingAArch32(EL2) then HDCR.HPMFZO else MDCR_EL2.HPMFZO;
            ovflws<hpmn-1:0> = Zeros(hpmn);
        else
            FZ = PMCR.FZO;
            if HaveEL(EL2) && hpmn < num_counters then
                ovflws<num_counters-1:hpmn> = Zeros(num_counters - hpmn);
        frozen = (FZ == '1') && !IsZero(ovflws);

    // PMCR.DP disables the cycle counter when event counting is prohibited
    if (prohibited || frozen) && idx == CYCLE_COUNTER_ID then
        enabled = enabled && (PMCR.DP == '0');
        // Otherwise whether event counting is prohibited does not affect the cycle counter
        prohibited = FALSE;
        frozen = FALSE;

    // If FEAT_PMUv3p5 is implemented, cycle counting can be prohibited.
    // This is not overridden by PMCR.DP.
    if HavePMUv3p5() && idx == CYCLE_COUNTER_ID then
        if HaveEL(EL3) && ss == SS_Secure then
            sccd = if ELUsingAArch32(EL3) then SDCR.SCCD else MDCR_EL3.SCCD;
            if sccd == '1' then prohibited = TRUE;
        if PSTATE.EL == EL2 && HDCR.HCCD == '1' then
            prohibited = TRUE;

    // Event counting can be filtered by the {P, U, NSK, NSU, NSH} bits
    filter = if idx == CYCLE_COUNTER_ID then PMCCFILTR else PMEVTYPER[idx];

    P   = filter<31>;
    U   = filter<30>;
    NSK = if HaveEL(EL3) then filter<29> else '0';
    NSU = if HaveEL(EL3) then filter<28> else '0';
    NSH = if HaveEL(EL2) then filter<27> else '0';

    ss = CurrentSecurityState();
    case PSTATE.EL of
        when EL0 filtered = if ss == SS_Secure then U == '1' else U != NSU;
        when EL1 filtered = if ss == SS_Secure then P == '1' else P != NSK;
        when EL2 filtered = NSH == '0';
        when EL3 filtered = P == '1';

    return !debug && enabled && !prohibited && !filtered && !frozen;
```

Library pseudocode for aarch32/debug/pmu/AArch32.GetNumEventCountersAccessible

```
// AArch32.GetNumEventCountersAccessible()
// =======================================
// Return the number of event counters that can be accessed at the current Exception level.

integer AArch32.GetNumEventCountersAccessible()
    integer n;
    integer total_counters = GetNumEventCounters();
    // Software can reserve some counters for EL2
    if PSTATE.EL IN {EL1, EL0} && EL2Enabled() then
        n = UInt(if !ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN);
        if n > total_counters || (!HaveFeatHPMN0() && n == 0) then
            (-, n) = ConstrainUnpredictableInteger(0, total_counters,
                                                   Unpredictable_PMUEVENTCOUNTER);
    else
        n = total_counters;

    return n;
```

Library pseudocode for aarch32/debug/pmu/AArch32.IncrementEventCounter

```
// AArch32.IncrementEventCounter()
// ===============================
// Increment the specified event counter by the specified amount.

AArch32.IncrementEventCounter(integer idx, integer increment)
    if HaveAArch64() then
        // Force the counter to be incremented as a 64-bit counter.
        AArch64.IncrementEventCounter(idx, increment);
        return;

    // In this model, event counters in an AArch32-only implementation are 32 bits and
    // the LP bits are RES0 in this model, even if FEAT_PMUv3p5 is implemented.
    integer old_value;
    integer new_value;
    integer ovflw;
    bit lp;
    old_value = UInt(PMEVCNTR[idx]);
    new_value = old_value + PMUCountValue(idx, increment);

    PMEVCNTR[idx] = new_value<31:0>;
    ovflw = 32;

    if old_value<64:ovflw> != new_value<64:ovflw> then
        PMOVSSET<idx> = '1';
        PMOVSR<idx> = '1';
        // Check for the CHAIN event from an even counter
        if idx<0> == '0' && idx + 1 < GetNumEventCounters() then
            PMUEvent(PMU_EVENT_CHAIN, 1, idx + 1);
```

Library pseudocode for aarch32/debug/pmu/AArch32.PMUCounterIsHyp

```
// AArch32.PMUCounterIsHyp
// =======================
// Returns TRUE if a counter is reserved for use by EL2, FALSE otherwise.

boolean AArch32.PMUCounterIsHyp(integer n)
    boolean resvd_for_el2;
    // Software can reserve some event counters for EL2
    if n != CYCLE_COUNTER_ID && HaveEL(EL2) then
        bits(5) hpmn_bits = if !ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN;
        resvd_for_el2 = n >= UInt(hpmn_bits);
        if UInt(hpmn_bits) > GetNumEventCounters() || (!HaveFeatHPMN0() && IsZero(hpmn_bits)) then
            resvd_for_el2 = boolean UNKNOWN;
    else
        resvd_for_el2 = FALSE;

    return resvd_for_el2;
```

Library pseudocode for aarch32/debug/pmu/AArch32.PMUCycle

```
// AArch32.PMUCycle()
// ==================
// Called at the end of each cycle to increment event counters and
// check for PMU overflow. In pseudocode, a cycle ends after the
// execution of the operational pseudocode.

AArch32.PMUCycle()
    if !HavePMUv3() then
        return;

    PMUEvent(PMU_EVENT_CPU_CYCLES);

    integer counters = GetNumEventCounters();
    if counters != 0 then
        for idx = 0 to counters - 1
            if AArch32.CountPMUEvents(idx) then
                accumulated = PMUEventAccumulator[idx];
                AArch32.IncrementEventCounter(idx, accumulated);
            PMUEventAccumulator[idx] = 0;

    integer old_value;
    integer new_value;
    integer ovflw;
    if (AArch32.CountPMUEvents(CYCLE_COUNTER_ID) &&
        (PMCR.LC == '1' || PMCR.D == '0' || HasElapsed64Cycles())) then
        old_value = UInt(PMCCNTR);
        new_value = old_value + 1;
        PMCCNTR = new_value<63:0>;

        ovflw = if PMCR.LC == '1' then 64 else 32;

        if old_value<64:ovflw> != new_value<64:ovflw> then
            PMOVSSET.C = '1';
            PMOVSR.C = '1';

    AArch32.CheckForPMUOverflow();
```

Library pseudocode for aarch32/debug/pmu/AArch32.PMUSwIncrement

```
// AArch32.PMUSwIncrement()
// ========================
// Generate PMU Events on a write to PMSWINC.

AArch32.PMUSwIncrement(bits(32) sw_incr)
    integer counters = AArch32.GetNumEventCountersAccessible();
    if counters != 0 then
        for idx = 0 to counters - 1
            if sw_incr<idx> == '1' then
                PMUEvent(PMU_EVENT_SW_INCR, 1, idx);
```

Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32.EnterHypModeInDebugState

```
// AArch32.EnterHypModeInDebugState()
// ==================================
// Take an exception in Debug state to Hyp mode.

AArch32.EnterHypModeInDebugState(ExceptionRecord exception)
    SynchronizeContext();
    assert HaveEL(EL2) && CurrentSecurityState() == SS_NonSecure && ELUsingAArch32(EL2);

    AArch32.ReportHypEntry(exception);
    AArch32.WriteMode(M32_Hyp);
    SPSR[] = bits(32) UNKNOWN;
    ELR_hyp = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1';                             // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    PSTATE.E = HSCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    EDSCR.ERR = '1';
    UpdateEDSCRFields();

    EndOfInstruction();
```

Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32.EnterModeInDebugState

```
// AArch32.EnterModeInDebugState()
// ===============================
// Take an exception in Debug state to a mode other than Monitor and Hyp mode.

AArch32.EnterModeInDebugState(bits(5) target_mode)
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR[] = bits(32) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1';                             // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() && SCTLR.SPAN == '0' then PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    EDSCR.ERR = '1';
    UpdateEDSCRFields();                        // Update EDSCR processor state flags.

    EndOfInstruction();
```

Library pseudocode for aarch32/debug/takeexceptiondbg/ AArch32.EnterMonitorModeInDebugState

```
// AArch32.EnterMonitorModeInDebugState()
// ======================================
// Take an exception in Debug state to Monitor mode.

AArch32.EnterMonitorModeInDebugState()
    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    from_secure = CurrentSecurityState() == SS_Secure;
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR[] = bits(32) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
    // In Debug state, the PE always execute T32 instructions when in AArch32 state, and
    // PSTATE.{SS,A,I,F} are not observable so behave as UNKNOWN.
    PSTATE.T = '1';                             // PSTATE.J is RES0
    PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTLR.SPAN == '0' then
            PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    EDSCR.ERR = '1';
    UpdateEDSCRFields();                        // Update EDSCR processor state flags.

    EndOfInstruction();
```

Library pseudocode for aarch32/debug/watchpoint/AArch32.WatchpointByteMatch

```
// AArch32.WatchpointByteMatch()
// =============================

boolean AArch32.WatchpointByteMatch(integer n,  bits(32) vaddress)

    integer top = 31;
    bottom = if DBGWVR[n]<2> == '1' then 2 else 3;           // Word or doubleword
    byte_select_match = (DBGWCR[n].BAS<UInt(vaddress<bottom-1:0>)> != '0');
    mask = UInt(DBGWCR[n].MASK);

    // If DBGWCR[n].MASK is non-zero value and DBGWCR[n].BAS is not set to '11111111', or
    // DBGWCR[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
    // UNPREDICTABLE.
    if mask > 0 && !IsOnes(DBGWCR[n].BAS) then
        byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPMASKANDBAS);
    else
        LSB = (DBGWCR[n].BAS AND NOT(DBGWCR[n].BAS - 1));  MSB = (DBGWCR[n].BAS + LSB);
        if !IsZero(MSB AND (MSB - 1)) then                   // Not contiguous
            byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPBASCONTIGUOUS);
            bottom = 3;                                      // For the whole doubleword

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        Constraint c;
        (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable_RESWPMASK);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
        case c of
            when Constraint_DISABLED  return FALSE;          // Disabled
            when Constraint_NONE      mask = 0;              // No masking
            // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

    boolean WVR_match;
    if mask > bottom then
        // If the DBGxVR<n>_EL1.RESS field bits are not a sign extension of the MSB
        // of DBGBVR<n>_EL1.VA, it is UNPREDICTABLE whether they appear to be
        // included in the match.
        if !IsOnes(DBGBVR_EL1[n]<63:top>) && !IsZero(DBGBVR_EL1[n]<63:top>) then
            if ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS) then
                top = 63;
        WVR_match = (vaddress<top:mask> == DBGWVR[n]<top:mask>);
        // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR[n]<mask-1:bottom>) then
            WVR_match = ConstrainUnpredictableBool(Unpredictable_WPMASKEDBITS);
    else
        WVR_match = vaddress<top:bottom> == DBGWVR[n]<top:bottom>;

    return WVR_match && byte_select_match;
```

Library pseudocode for aarch32/debug/watchpoint/AArch32.WatchpointMatch

```
// AArch32.WatchpointMatch()
// =========================
// Watchpoint matching in an AArch32 translation regime.

boolean AArch32.WatchpointMatch(integer n, bits(32) vaddress, integer size, boolean ispriv,
                                AccType acctype, boolean iswrite)
    assert ELUsingAArch32(S1TranslationRegime());
    assert n < NumWatchpointsImplemented();

    // "ispriv" is:
    //  * FALSE for all loads, stores, and atomic operations executed at EL0.
    //  * FALSE if the access is unprivileged.
    //  * TRUE for all other loads, stores, and atomic operations.

    enabled = DBGWCR[n].E == '1';
    linked = DBGWCR[n].WT == '1';
    isbreakpnt = FALSE;

    state_match = AArch32.StateMatch(DBGWCR[n].SSC, DBGWCR[n].HMC, DBGWCR[n].PAC,
                                     linked, DBGWCR[n].LBN, isbreakpnt,  ispriv);
    ls_match = FALSE;
    if iswrite then
        ls_match = (DBGWCR[n].LSC<1> != '0');
    else
        ls_match = (DBGWCR[n].LSC<0> != '0');

    value_match = FALSE;
    for byte = 0 to size - 1
        value_match = value_match || AArch32.WatchpointByteMatch(n,  vaddress + byte);

    return value_match && state_match && ls_match && enabled;
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.Abort

```
// AArch32.Abort()
// ===============
// Abort and Debug exception handling in an AArch32 translation regime.

AArch32.Abort(bits(32) vaddress, FaultRecord fault)

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);

    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = (HCR_EL2.TGE == '1' || IsSecondStage(fault) ||
                            (HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault)) ||
                            (IsDebugException(fault) && MDCR_EL2.TDE == '1'));

    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.EA == '1' && IsExternalAbort(fault);

    if route_to_aarch64 then
        AArch64.Abort(ZeroExtend(vaddress,  64), fault);
    elsif fault.acctype == AccType_IFETCH then
        AArch32.TakePrefetchAbortException(vaddress, fault);
    else
        AArch32.TakeDataAbortException(vaddress, fault);
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.AbortSyndrome

```
// AArch32.AbortSyndrome()
// =======================
// Creates an exception syndrome record for Abort  exceptions taken to Hyp mode
// from an AArch32 translation regime.

ExceptionRecord AArch32.AbortSyndrome(Exception exceptype, FaultRecord fault, bits(32) vaddress)
    exception = ExceptionSyndrome(exceptype);

    d_side = exceptype == Exception_DataAbort;

    exception.syndrome = AArch32.FaultSyndrome(d_side, fault);
    exception.vaddress = ZeroExtend(vaddress, 64);
    if IPAValid(fault) then
        exception.ipavalid = TRUE;
        exception.NS = if fault.ipaddress.paspace == PAS_NonSecure then '1' else '0';
        exception.ipaddress = ZeroExtend(fault.ipaddress.address,  52);
    else
        exception.ipavalid = FALSE;

    return exception;
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.CheckPCAlignment

```
// AArch32.CheckPCAlignment()
// ==========================

AArch32.CheckPCAlignment()

    bits(32) pc = ThisInstrAddr(32);
    if (CurrentInstrSet() == InstrSet_A32 && pc<1> == '1') || pc<0> == '1' then
        if AArch32.GeneralExceptionsToAArch64() then AArch64.PCAlignmentFault();

        // Generate an Alignment fault Prefetch Abort exception
        vaddress = pc;
        acctype = AccType_IFETCH;
        iswrite = FALSE;
        secondstage = FALSE;
        AArch32.Abort(vaddress, AlignmentFault(acctype, iswrite, secondstage));
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.ReportDataAbort

```
// AArch32.ReportDataAbort()
// =========================
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportDataAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)
    long_format = FALSE;
    if route_to_monitor && CurrentSecurityState() != SS_Secure then
        long_format = ((TTBCR_S.EAE == '1') ||
                       (IsExternalSyncAbort(fault) && ((PSTATE.EL == EL2 || TTBCR.EAE == '1') ||
                        (fault.secondstage && boolean IMPLEMENTATION_DEFINED "Stage 2 synchronous Externa
    else
        long_format = TTBCR.EAE == '1';
    d_side = TRUE;
    bits(32) syndrome;
    if long_format then
        syndrome = AArch32.FaultStatusLD(d_side, fault);
    else
        syndrome = AArch32.FaultStatusSD(d_side, fault);

    if fault.acctype == AccType_IC then
        bits(32) i_syndrome;
        if (!long_format &&
            boolean IMPLEMENTATION_DEFINED "Report I-cache maintenance fault in IFSR") then
            i_syndrome = syndrome;
            syndrome<10,3:0> = EncodeSDFSC(Fault_ICacheMaint, 1);
        else
            i_syndrome = bits(32) UNKNOWN;
        if route_to_monitor then
            IFSR_S = i_syndrome;
        else
            IFSR = i_syndrome;

    if route_to_monitor then
        DFSR_S = syndrome;
        DFAR_S = vaddress;
    else
        DFSR = syndrome;
        DFAR = vaddress;

    return;
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.ReportPrefetchAbort

```
// AArch32.ReportPrefetchAbort()
// =============================
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportPrefetchAbort(boolean route_to_monitor, FaultRecord fault, bits(32) vaddress)
    // The encoding used in the IFSR can be Long-descriptor format or Short-descriptor format.
    // Normally, the current translation table format determines the format. For an abort from
    // Non-secure state to Monitor mode, the IFSR uses the Long-descriptor format if any of the
    // following applies:
    // * The Secure TTBCR.EAE is set to 1.
    // * It is taken from Hyp mode.
    // * It is taken from EL1 or EL0, and the Non-secure TTBCR.EAE is set to 1.
    long_format = FALSE;
    if route_to_monitor && CurrentSecurityState() != SS_Secure then
        long_format = TTBCR_S.EAE == '1' || PSTATE.EL == EL2 || TTBCR.EAE == '1';
    else
        long_format = TTBCR.EAE == '1';

    d_side = FALSE;
    bits(32) fsr;
    if long_format then
        fsr = AArch32.FaultStatusLD(d_side, fault);
    else
        fsr = AArch32.FaultStatusSD(d_side, fault);

    if route_to_monitor then
        IFSR_S = fsr;
        IFAR_S = vaddress;
    else
        IFSR = fsr;
        IFAR = vaddress;

    return;
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.TakeDataAbortException

```
// AArch32.TakeDataAbortException()
// ================================

AArch32.TakeDataAbortException(bits(32) vaddress, FaultRecord fault)
    route_to_monitor = HaveEL(EL3) && SCR.EA == '1' && IsExternalAbort(fault);
    route_to_hyp = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
                    (HCR.TGE == '1' ||
                     (HaveRASExt() && HCR2.TEA == '1' && IsExternalAbort(fault)) ||
                     (IsDebugException(fault) && HDCR.TDE == '1') ||
                     IsSecondStage(fault)));

    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x10;
    lr_offset = 8;

    if IsDebugException(fault) then DBGDSCRext.MOE = fault.debugmoe;
    if route_to_monitor then
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        exception = AArch32.AbortSyndrome(Exception_DataAbort, fault, vaddress);
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.TakePrefetchAbortException

```
// AArch32.TakePrefetchAbortException()
// ====================================

AArch32.TakePrefetchAbortException(bits(32) vaddress, FaultRecord fault)
    route_to_monitor = HaveEL(EL3) && SCR.EA == '1' && IsExternalAbort(fault);
    route_to_hyp = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
                    (HCR.TGE == '1' ||
                     (HaveRASExt() && HCR2.TEA == '1' && IsExternalAbort(fault)) ||
                     (IsDebugException(fault) && HDCR.TDE == '1') ||
                     IsSecondStage(fault)));

    ExceptionRecord exception;
    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x0C;

    lr_offset = 4;

    if IsDebugException(fault) then DBGDSCRext.MOE = fault.debugmoe;
    if route_to_monitor then
        AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        if fault.statuscode == Fault_Alignment then             // PC Alignment fault
            exception = ExceptionSyndrome(Exception_PCAlignment);
            exception.vaddress = ThisInstrAddr(64);
        else
            exception = AArch32.AbortSyndrome(Exception_InstructionAbort, fault, vaddress);
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/async/AArch32.TakePhysicalFIQException

```
// AArch32.TakePhysicalFIQException()
// ==================================

AArch32.TakePhysicalFIQException()

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = HCR_EL2.TGE == '1' || (HCR_EL2.FMO == '1' && !IsInHost());

    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.FIQ == '1';

    if route_to_aarch64 then AArch64.TakePhysicalFIQException();
    route_to_monitor = HaveEL(EL3) && SCR.FIQ == '1';
    route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR.TGE == '1' || HCR.FMO == '1'));
    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x1C;
    lr_offset = 4;
    if route_to_monitor then
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        exception = ExceptionSyndrome(Exception_FIQ);
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/async/AArch32.TakePhysicalIRQException

```
// AArch32.TakePhysicalIRQException()
// ==================================
// Take an enabled physical IRQ exception.

AArch32.TakePhysicalIRQException()

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = HCR_EL2.TGE == '1' || (HCR_EL2.IMO == '1' && !IsInHost());
    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.IRQ == '1';

    if route_to_aarch64 then AArch64.TakePhysicalIRQException();

    route_to_monitor = HaveEL(EL3) && SCR.IRQ == '1';
    route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR.TGE == '1' || HCR.IMO == '1'));
    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x18;
    lr_offset = 4;
    if route_to_monitor then
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_hyp then
        exception = ExceptionSyndrome(Exception_IRQ);
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/async/AArch32.TakePhysicalSErrorException

```
// AArch32.TakePhysicalSErrorException()
// =====================================

AArch32.TakePhysicalSErrorException(boolean parity, bit extflag, bits(2) pe_error_state,
                                    bits(25) full_syndrome)
    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);

    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = (HCR_EL2.TGE == '1' || (!IsInHost() && HCR_EL2.AMO == '1'));
    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.EA == '1';

    if route_to_aarch64 then
        AArch64.TakePhysicalSErrorException(full_syndrome);

    route_to_monitor = HaveEL(EL3) && SCR.EA == '1';
    route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR.TGE == '1' || HCR.AMO == '1'));
    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x10;
    lr_offset = 8;

    bits(2) target_el;
    if route_to_monitor then
        target_el = EL3;
    elsif PSTATE.EL == EL2 || route_to_hyp then
        target_el = EL2;
    else
        target_el = EL1;

    if IsSErrorEdgeTriggered(target_el, full_syndrome) then
        ClearPendingPhysicalSError();

    fault = AsyncExternalAbort(parity, pe_error_state, extflag);
    vaddress = bits(32) UNKNOWN;

    case target_el of
        when EL3
            AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
            AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
        when EL2
            exception = AArch32.AbortSyndrome(Exception_DataAbort, fault, vaddress);
            if PSTATE.EL == EL2 then
                AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
            else
                AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
        when EL1
            AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
            AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
        otherwise
            Unreachable();
```

Library pseudocode for aarch32/exceptions/async/AArch32.TakeVirtualFIQException

```
// AArch32.TakeVirtualFIQException()
// =================================

AArch32.TakeVirtualFIQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    if ELUsingAArch32(EL2) then  // Virtual IRQ enabled if TGE==0 and FMO==1
        assert HCR.TGE == '0' && HCR.FMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1';
    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualFIQException();

    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x1C;
    lr_offset = 4;

    AArch32.EnterMode(M32_FIQ, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/async/AArch32.TakeVirtualIRQException

```
// AArch32.TakeVirtualIRQException()
// =================================

AArch32.TakeVirtualIRQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    if ELUsingAArch32(EL2) then   // Virtual IRQs enabled if TGE==0 and IMO==1
        assert HCR.TGE == '0' && HCR.IMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1';

    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualIRQException();

    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x18;
    lr_offset = 4;

    AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/async/AArch32.TakeVirtualSErrorException

```
// AArch32.TakeVirtualSErrorException()
// ====================================

AArch32.TakeVirtualSErrorException(bit extflag, bits(2) pe_error_state, bits(25) full_syndrome)

    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    if ELUsingAArch32(EL2) then   // Virtual SError enabled if TGE==0 and AMO==1
        assert HCR.TGE == '0' && HCR.AMO == '1';
    else
        assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';
    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualSErrorException(full_syndrome);

    route_to_monitor = FALSE;

    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x10;
    lr_offset = 8;

    vaddress = bits(32) UNKNOWN;
    parity = FALSE;
    FaultRecord fault;
    if HaveRASExt() then
        if ELUsingAArch32(EL2) then
            fault = AsyncExternalAbort(FALSE, VDFSR.AET, VDFSR.ExT);
        else
            fault = AsyncExternalAbort(FALSE, VSESR_EL2.AET, VSESR_EL2.ExT);
    else
        fault = AsyncExternalAbort(parity, pe_error_state, extflag);

    ClearPendingVirtualSError();
    AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
    AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/debug/AArch32.SoftwareBreakpoint

```
// AArch32.SoftwareBreakpoint()
// ============================

AArch32.SoftwareBreakpoint(bits(16) immediate)

    if (EL2Enabled() && !ELUsingAArch32(EL2) &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1')) || !ELUsingAArch32(EL1) then
        AArch64.SoftwareBreakpoint(immediate);
    vaddress = bits(32) UNKNOWN;
    acctype = AccType_IFETCH;           // Take as a Prefetch Abort
    iswrite = FALSE;
    entry = DebugException_BKPT;

    fault = AArch32.DebugFault(acctype, iswrite, entry);
    AArch32.Abort(vaddress, fault);
```

Library pseudocode for aarch32/exceptions/debug/DebugException

```
constant bits(4) DebugException_Breakpoint  = '0001';
constant bits(4) DebugException_BKPT        = '0011';
constant bits(4) DebugException_VectorCatch = '0101';
constant bits(4) DebugException_Watchpoint  = '1010';
```

Library pseudocode for aarch32/exceptions/exceptions/ AArch32.CheckAdvSIMDOrFPRegisterTraps

```
// AArch32.CheckAdvSIMDOrFPRegisterTraps()
// =======================================
// Check if an instruction that accesses an Advanced SIMD and
// floating-point System register is trapped by an appropriate HCR.TIDx
// ID group trap control.

AArch32.CheckAdvSIMDOrFPRegisterTraps(bits(4) reg)

    if PSTATE.EL == EL1 && EL2Enabled() then
        tid0 = if ELUsingAArch32(EL2) then HCR.TID0 else HCR_EL2.TID0;
        tid3 = if ELUsingAArch32(EL2) then HCR.TID3 else HCR_EL2.TID3;

        if (tid0 == '1' && reg == '0000')                           // FPSID
          || (tid3 == '1' && reg IN {'0101', '0110', '0111'}) then  // MVFRx
            if ELUsingAArch32(EL2) then
                AArch32.SystemAccessTrap(M32_Hyp, 0x8);             // Exception_AdvSIMDFPAccessTrap
            else
                AArch64.AArch32SystemAccessTrap(EL2, 0x8);          // Exception_AdvSIMDFPAccessTrap
```

Library pseudocode for aarch32/exceptions/exceptions/AArch32.ExceptionClass

```
// AArch32.ExceptionClass()
// ========================
// Returns the Exception Class and Instruction Length fields to be reported in HSR

(integer,bit) AArch32.ExceptionClass(Exception exceptype)

    il_is_valid = TRUE;
    integer ec;
    case exceptype of
        when Exception_Uncategorized        ec = 0x00; il_is_valid = FALSE;
        when Exception_WFxTrap              ec = 0x01;
        when Exception_CP15RTTrap           ec = 0x03;
        when Exception_CP15RRTTrap          ec = 0x04;
        when Exception_CP14RTTrap           ec = 0x05;
        when Exception_CP14DTTrap           ec = 0x06;
        when Exception_AdvSIMDFPAccessTrap  ec = 0x07;
        when Exception_FPIDTrap             ec = 0x08;
        when Exception_PACTrap              ec = 0x09;
        when Exception_TSTARTAccessTrap     ec = 0x1B;
        when Exception_GPC                  ec = 0x1E;
        when Exception_CP14RRTTrap          ec = 0x0C;
        when Exception_BranchTarget         ec = 0x0D;
        when Exception_IllegalState         ec = 0x0E; il_is_valid = FALSE;
        when Exception_SupervisorCall       ec = 0x11;
        when Exception_HypervisorCall       ec = 0x12;
        when Exception_MonitorCall          ec = 0x13;
        when Exception_InstructionAbort     ec = 0x20; il_is_valid = FALSE;
        when Exception_PCAlignment          ec = 0x22; il_is_valid = FALSE;
        when Exception_DataAbort            ec = 0x24;
        when Exception_NV2DataAbort         ec = 0x25;
        when Exception_FPTrappedException   ec = 0x28;
        otherwise                           Unreachable();

    if ec IN {0x20,0x24} && PSTATE.EL == EL2 then
        ec = ec + 1;
    bit il;
    if il_is_valid then
        il = if ThisInstrLength() == 32 then '1' else '0';
    else
        il = '1';

    return (ec,il);
```

Library pseudocode for aarch32/exceptions/exceptions/AArch32.GeneralExceptionsToAArch64

```
// AArch32.GeneralExceptionsToAArch64()
// ====================================
// Returns TRUE if exceptions normally routed to EL1 are being handled at an Exception
// level using AArch64, because either EL1 is using AArch64 or TGE is in force and EL2
// is using AArch64.

boolean AArch32.GeneralExceptionsToAArch64()
    return ((PSTATE.EL == EL0 && !ELUsingAArch32(EL1)) ||
            (EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1'));
```

Library pseudocode for aarch32/exceptions/exceptions/AArch32.ReportHypEntry

```
// AArch32.ReportHypEntry()
// ========================
// Report syndrome information to Hyp mode registers.

AArch32.ReportHypEntry(ExceptionRecord exception)

    Exception exceptype = exception.exceptype;

    (ec,il) = AArch32.ExceptionClass(exceptype);
    iss = exception.syndrome;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';

    HSR = ec<5:0>:il:iss;

    if exceptype IN {Exception_InstructionAbort, Exception_PCAlignment} then
        HIFAR = exception.vaddress<31:0>;
        HDFAR = bits(32) UNKNOWN;
    elsif exceptype == Exception_DataAbort then
        HIFAR = bits(32) UNKNOWN;
        HDFAR = exception.vaddress<31:0>;

    if exception.ipavalid then
        HPFAR<31:4> = exception.ipaddress<39:12>;
    else
        HPFAR<31:4> = bits(28) UNKNOWN;

    return;
```

Library pseudocode for aarch32/exceptions/exceptions/AArch32.ResetControlRegisters

```
// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.
AArch32.ResetControlRegisters(boolean cold_reset);
```

Library pseudocode for aarch32/exceptions/exceptions/AArch32.TakeReset

```
// AArch32.TakeReset()
// ===================
// Reset into AArch32 state

AArch32.TakeReset(boolean cold_reset)
    assert !HaveAArch64();

    // Enter the highest implemented Exception level in AArch32 state
    if HaveEL(EL3) then
        AArch32.WriteMode(M32_Svc);
        SCR.NS = '0';                     // Secure state
    elsif HaveEL(EL2) then
        AArch32.WriteMode(M32_Hyp);
    else
        AArch32.WriteMode(M32_Svc);

    // Reset System registers in the coproc=0b111x encoding space and other system components
    AArch32.ResetControlRegisters(cold_reset);
    FPEXC.EN = '0';

    // Reset all other PSTATE fields, including instruction set and endianness according to the
    // SCTLR values produced by the above call to ResetControlRegisters()
    PSTATE.<A,I,F> = '111';       // All asynchronous exceptions masked
    PSTATE.IT = '00000000';       // IT block state reset
    if HaveEL(EL2) && !HaveEL(EL3) then
        PSTATE.T = HSCTLR.TE;     // Instruction set: TE=0: A32, TE=1: T32. PSTATE.J is RES0.
        PSTATE.E = HSCTLR.EE;     // Endianness: EE=0: little-endian, EE=1: big-endian
    else
        PSTATE.T = SCTLR.TE;      // Instruction set: TE=0: A32, TE=1: T32. PSTATE.J is RES0.
        PSTATE.E = SCTLR.EE;      // Endianness: EE=0: little-endian, EE=1: big-endian
    PSTATE.IL = '0';              // Clear Illegal Execution state bit

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // R14 or ELR_hyp and SPSR have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch32.ResetGeneralRegisters();
    AArch32.ResetSIMDFPRegisters();
    AArch32.ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);

    bits(32) rv;                      // IMPLEMENTATION DEFINED reset vector

    if HaveEL(EL3) then
        if MVBAR<0> == '1' then           // Reset vector in MVBAR
            rv = MVBAR<31:1>:'0';
        else
            rv = bits(32) IMPLEMENTATION_DEFINED "reset vector address";
    else
        rv = RVBAR<31:1>:'0';

    // The reset vector must be correctly aligned
    assert rv<0> == '0' && (PSTATE.T == '1' || rv<1> == '0');

    boolean branch_conditional = FALSE;
    BranchTo(rv, BranchType_RESET, branch_conditional);
```

Library pseudocode for aarch32/exceptions/exceptions/ExcVectorBase

```
// ExcVectorBase()
// ===============

bits(32) ExcVectorBase()
    if SCTLR.V == '1' then  // Hivecs selected, base = 0xFFFF0000
        return Ones(16):Zeros(16);
    else
        return VBAR<31:5>:Zeros(5);
```

Library pseudocode for aarch32/exceptions/ieeefp/AArch32.FPTrappedException

```
// AArch32.FPTrappedException()
// ============================

AArch32.FPTrappedException(bits(8) accumulated_exceptions)
    if AArch32.GeneralExceptionsToAArch64() then
        is_ase = FALSE;
        element = 0;
        AArch64.FPTrappedException(is_ase, accumulated_exceptions);
    FPEXC.DEX    = '1';
    FPEXC.TFV    = '1';
    FPEXC<7,4:0> = accumulated_exceptions<7,4:0>;                 // IDF,IXF,UFF,OFF,DZF,IOF
    FPEXC<10:8>  = '111';                                         // VECITR is RES1

    AArch32.TakeUndefInstrException();
```

Library pseudocode for aarch32/exceptions/syscalls/AArch32.CallHypervisor

```
// AArch32.CallHypervisor()
// ========================
// Performs a HVC call

AArch32.CallHypervisor(bits(16) immediate)
    assert HaveEL(EL2);

    if !ELUsingAArch32(EL2) then
        AArch64.CallHypervisor(immediate);
    else
        AArch32.TakeHVCException(immediate);
```

Library pseudocode for aarch32/exceptions/syscalls/AArch32.CallSupervisor

```
// AArch32.CallSupervisor()
// ========================
// Calls the Supervisor

AArch32.CallSupervisor(bits(16) immediate_in)
    bits(16) immediate = immediate_in;
    if AArch32.CurrentCond() != '1110' then
        immediate = bits(16) UNKNOWN;
    if AArch32.GeneralExceptionsToAArch64() then
        AArch64.CallSupervisor(immediate);
    else
        AArch32.TakeSVCException(immediate);
```

Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeHVCException

```
// AArch32.TakeHVCException()
// ==========================

AArch32.TakeHVCException(bits(16) immediate)
    assert HaveEL(EL2) && ELUsingAArch32(EL2);

    AArch32.ITAdvance();
    SSAdvance();
    bits(32) preferred_exception_return = NextInstrAddr(32);
    vect_offset = 0x08;

    exception = ExceptionSyndrome(Exception_HypervisorCall);
    exception.syndrome<15:0> = immediate;

    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
```

Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeSMCException

```
// AArch32.TakeSMCException()
// ==========================

AArch32.TakeSMCException()
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    AArch32.ITAdvance();
    SSAdvance();
    bits(32) preferred_exception_return = NextInstrAddr(32);
    vect_offset = 0x08;
    lr_offset = 0;

    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeSVCException

```
// AArch32.TakeSVCException()
// ==========================

AArch32.TakeSVCException(bits(16) immediate)

    AArch32.ITAdvance();
    SSAdvance();
    route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';

    bits(32) preferred_exception_return = NextInstrAddr(32);
    vect_offset = 0x08;
    lr_offset = 0;

    if PSTATE.EL == EL2 || route_to_hyp then
        exception = ExceptionSyndrome(Exception_SupervisorCall);
        exception.syndrome<15:0> = immediate;
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.EnterMode(M32_Svc, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/takeexception/AArch32.EnterHypMode

```
// AArch32.EnterHypMode()
// ======================
// Take an exception to Hyp mode.

AArch32.EnterHypMode(ExceptionRecord exception, bits(32) preferred_exception_return,
                     integer vect_offset)
    SynchronizeContext();
    assert HaveEL(EL2) && CurrentSecurityState() == SS_NonSecure && ELUsingAArch32(EL2);

    bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState, 32);
    if !(exception.exceptype IN {Exception_IRQ, Exception_FIQ}) then
        AArch32.ReportHypEntry(exception);
    AArch32.WriteMode(M32_Hyp);
    SPSR[] = spsr;
    ELR_hyp = preferred_exception_return;
    PSTATE.T = HSCTLR.TE;                       // PSTATE.J is RES0
    PSTATE.SS = '0';
    if !HaveEL(EL3) || SCR_GEN[].EA == '0' then PSTATE.A = '1';
    if !HaveEL(EL3) || SCR_GEN[].IRQ == '0' then PSTATE.I = '1';
    if !HaveEL(EL3) || SCR_GEN[].FIQ == '0' then PSTATE.F = '1';
    PSTATE.E = HSCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HaveSSBSExt() then PSTATE.SSBS = HSCTLR.DSSBS;
    boolean branch_conditional = FALSE;
    BranchTo(HVBAR<31:5>:vect_offset<4:0>, BranchType_EXCEPTION, branch_conditional);

    CheckExceptionCatch(TRUE);                  // Check for debug event on exception entry

    EndOfInstruction();
```

Library pseudocode for aarch32/exceptions/takeexception/AArch32.EnterMode

```
// AArch32.EnterMode()
// ===================
// Take an exception to a mode other than Monitor and Hyp mode.

AArch32.EnterMode(bits(5) target_mode, bits(32) preferred_exception_return, integer lr_offset,
                  integer vect_offset)
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState, 32);
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR[] = spsr;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTLR.TE;                        // PSTATE.J is RES0
    PSTATE.SS = '0';
    if target_mode == M32_FIQ then
        PSTATE.<A,I,F> = '111';
    elsif target_mode IN {M32_Abort, M32_IRQ} then
        PSTATE.<A,I> = '11';
    else
        PSTATE.I = '1';
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() && SCTLR.SPAN == '0' then PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = SCTLR.DSSBS;
    boolean branch_conditional = FALSE;
    BranchTo(ExcVectorBase()<31:5>:vect_offset<4:0>, BranchType_EXCEPTION, branch_conditional);

    CheckExceptionCatch(TRUE);                  // Check for debug event on exception entry

    EndOfInstruction();
```

Library pseudocode for aarch32/exceptions/takeexception/AArch32.EnterMonitorMode

```
// AArch32.EnterMonitorMode()
// ==========================
// Take an exception to Monitor mode.

AArch32.EnterMonitorMode(bits(32) preferred_exception_return, integer lr_offset,
                         integer vect_offset)
    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    from_secure = CurrentSecurityState() == SS_Secure;
    bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState, 32);
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR[] = spsr;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTLR.TE;                        // PSTATE.J is RES0
    PSTATE.SS = '0';
    PSTATE.<A,I,F> = '111';
    PSTATE.E = SCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTLR.SPAN == '0' then
            PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = SCTLR.DSSBS;
    boolean branch_conditional = FALSE;
    BranchTo(MVBAR<31:5>:vect_offset<4:0>, BranchType_EXCEPTION, branch_conditional);

    CheckExceptionCatch(TRUE);                  // Check for debug event on exception entry

    EndOfInstruction();
```

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckAdvSIMDOrFPEnabled

```
// AArch32.CheckAdvSIMDOrFPEnabled()
// =================================
// Check against CPACR, FPEXC, HCPTR, NSACR, and CPTR_EL3.

AArch32.CheckAdvSIMDOrFPEnabled(boolean fpexc_check, boolean advsimd)
    if PSTATE.EL == EL0 && (!EL2Enabled() || (!ELUsingAArch32(EL2) && HCR_EL2.TGE == '0')) && !ELUsingAAr
        // The PE behaves as if FPEXC.EN is 1
        AArch64.CheckFPEnabled();
        AArch64.CheckFPAdvSIMDEnabled();
    elsif PSTATE.EL == EL0 && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' && !ELUsingAArch
        if fpexc_check && HCR_EL2.RW == '0' then
            fpexc_en = bits(1) IMPLEMENTATION_DEFINED "FPEXC.EN value when TGE==1 and RW==0";
            if fpexc_en == '0' then UNDEFINED;
        AArch64.CheckFPEnabled();
    else
        cpacr_asedis = CPACR.ASEDIS;
        cpacr_cp10 = CPACR.cp10;

        if HaveEL(EL3) && ELUsingAArch32(EL3) && CurrentSecurityState() == SS_NonSecure then
            // Check if access disabled in NSACR
            if NSACR.NSASEDIS == '1' then cpacr_asedis = '1';
            if NSACR.cp10 == '0' then cpacr_cp10 = '00';

        if PSTATE.EL != EL2 then
            // Check if Advanced SIMD disabled in CPACR
            if advsimd && cpacr_asedis == '1' then UNDEFINED;

            // Check if access disabled in CPACR
            boolean disabled;
            case cpacr_cp10 of
                when '00' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0;
                when '10' disabled = ConstrainUnpredictableBool(Unpredictable_RESCPACR);
                when '11' disabled = FALSE;
            if disabled then UNDEFINED;

        // If required, check FPEXC enabled bit.
        if fpexc_check && FPEXC.EN == '0' then UNDEFINED;

        AArch32.CheckFPAdvSIMDTrap(advsimd);    // Also check against HCPTR and CPTR_EL3
```

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckFPAdvSIMDTrap

```
// AArch32.CheckFPAdvSIMDTrap()
// ============================
// Check against CPTR_EL2 and CPTR_EL3.

AArch32.CheckFPAdvSIMDTrap(boolean advsimd)
    if EL2Enabled() && !ELUsingAArch32(EL2) then
        AArch64.CheckFPAdvSIMDTrap();
    else
        if (HaveEL(EL3) && !ELUsingAArch32(EL3) &&
            CPTR_EL3.TFP == '1' && EL3SDDTrapPriority()) then
            UNDEFINED;

        ss = CurrentSecurityState();
        if HaveEL(EL2) && ss != SS_Secure then
            hcptr_tase = HCPTR.TASE;
            hcptr_cp10 = HCPTR.TCP10;

            if HaveEL(EL3) && ELUsingAArch32(EL3) then
                // Check if access disabled in NSACR
                if NSACR.NSASEDIS == '1' then hcptr_tase = '1';
                if NSACR.cp10 == '0' then hcptr_cp10 = '1';

            // Check if access disabled in HCPTR
            if (advsimd && hcptr_tase == '1') || hcptr_cp10 == '1' then
                exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
                exception.syndrome<24:20> = ConditionSyndrome();

                if advsimd then
                    exception.syndrome<5> = '1';
                else
                    exception.syndrome<5> = '0';
                    exception.syndrome<3:0> = '1010';         // coproc field, always 0xA

                if PSTATE.EL == EL2 then
                    AArch32.TakeUndefInstrException(exception);
                else
                    AArch32.TakeHypTrapException(exception);

        if HaveEL(EL3) && !ELUsingAArch32(EL3) then
            // Check if access disabled in CPTR_EL3
            if CPTR_EL3.TFP == '1' then
                if Halted() && EDSCR.SDD == '1' then
                    UNDEFINED;
                else
                    AArch64.AdvSIMDFPAccessTrap(EL3);
```

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckForSMCUndefOrTrap

```
// AArch32.CheckForSMCUndefOrTrap()
// ================================
// Check for UNDEFINED or trap on SMC instruction

AArch32.CheckForSMCUndefOrTrap()
    if !HaveEL(EL3) || PSTATE.EL == EL0 then
        UNDEFINED;

    if EL2Enabled() && !ELUsingAArch32(EL2) then
        AArch64.CheckForSMCUndefOrTrap(Zeros(16));
    else
        route_to_hyp = EL2Enabled() && PSTATE.EL == EL1 && HCR.TSC == '1';
        if route_to_hyp then
            exception = ExceptionSyndrome(Exception_MonitorCall);
            AArch32.TakeHypTrapException(exception);
```

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckForSVCTrap

```
// AArch32.CheckForSVCTrap()
// =========================
// Check for trap on SVC instruction

AArch32.CheckForSVCTrap(bits(16) immediate)
    if HaveFGTExt() then
        route_to_el2 = FALSE;
        if PSTATE.EL == EL0 then
            route_to_el2 = (!ELUsingAArch32(EL1) && EL2Enabled() && HFGITR_EL2.SVC_EL0 == '1' &&
                            (HCR_EL2.<E2H, TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')));

        if route_to_el2 then
            exception = ExceptionSyndrome(Exception_SupervisorCall);
            exception.syndrome<15:0> = immediate;
            exception.trappedsyscallinst = TRUE;
            bits(64) preferred_exception_return = ThisInstrAddr(64);
            vect_offset = 0x0;

            AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckForWFxTrap

```
// AArch32.CheckForWFxTrap()
// =========================
// Check for trap on WFE or WFI instruction

AArch32.CheckForWFxTrap(bits(2) target_el, WFxType wfxtype)
    assert HaveEL(target_el);

    // Check for routing to AArch64
    if !ELUsingAArch32(target_el) then
        AArch64.CheckForWFxTrap(target_el, wfxtype);
        return;

    boolean is_wfe = wfxtype == WFxType_WFE;
    boolean trap;
    case target_el of
        when EL1
            trap = (if is_wfe then SCTLR.nTWE else SCTLR.nTWI) == '0';
        when EL2
            trap = (if is_wfe then HCR.TWE else HCR.TWI) == '1';
        when EL3
            trap = (if is_wfe then SCR.TWE else SCR.TWI) == '1';

    if trap then
        if target_el == EL1 && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.WFxTrap(wfxtype, target_el);

        if target_el == EL3 then
            AArch32.TakeMonitorTrapException();
        elsif target_el == EL2 then
            exception = ExceptionSyndrome(Exception_WFxTrap);
            exception.syndrome<24:20> = ConditionSyndrome();

            case wfxtype of
                when WFxType_WFI
                    exception.syndrome<0> = '0';
                when WFxType_WFE
                    exception.syndrome<0> = '1';

            AArch32.TakeHypTrapException(exception);
        else
            AArch32.TakeUndefInstrException();
```

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckITEnabled

```
// AArch32.CheckITEnabled()
// ========================
// Check whether the T32 IT instruction is disabled.

AArch32.CheckITEnabled(bits(4) mask)
    bit it_disabled;
    if PSTATE.EL == EL2 then
        it_disabled = HSCTLR.ITD;
    else
        it_disabled = (if ELUsingAArch32(EL1) then SCTLR.ITD else SCTLR[].ITD);
    if it_disabled == '1' then
        if mask != '1000' then UNDEFINED;

        // Otherwise whether the IT block is allowed depends on hw1 of the next instruction.
        next_instr = AArch32.MemSingle[NextInstrAddr(32), 2, AccType_IFETCH, TRUE];

        if next_instr IN {'11xxxxxxxxxxxxxx', '1011xxxxxxxxxxxx', '10100xxxxxxxxxxx',
                          '01001xxxxxxxxxxx', '010001xxx1111xxx', '010001xx1xxxx111'} then
            // It is IMPLEMENTATION DEFINED whether the Undefined Instruction exception is
            // taken on the IT instruction or the next instruction. This is not reflected in
            // the pseudocode, which always takes the exception on the IT instruction. This
            // also does not take into account cases where the next instruction is UNPREDICTABLE.
            UNDEFINED;

    return;
```

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckIllegalState

```
// AArch32.CheckIllegalState()
// ===========================
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

AArch32.CheckIllegalState()
    if AArch32.GeneralExceptionsToAArch64() then
        AArch64.CheckIllegalState();
    elsif PSTATE.IL == '1' then
        route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';

        bits(32) preferred_exception_return = ThisInstrAddr(32);
        vect_offset = 0x04;

        if PSTATE.EL == EL2 || route_to_hyp then
            exception = ExceptionSyndrome(Exception_IllegalState);
            if PSTATE.EL == EL2 then
                AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
            else
                AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
        else
            AArch32.TakeUndefInstrException();
```

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckSETENDEnabled

```
// AArch32.CheckSETENDEnabled()
// ============================
// Check whether the AArch32 SETEND instruction is disabled.

AArch32.CheckSETENDEnabled()
    bit setend_disabled;
    if PSTATE.EL == EL2 then
        setend_disabled = HSCTLR.SED;
    else
        setend_disabled = (if ELUsingAArch32(EL1) then SCTLR.SED else SCTLR[].SED);
    if setend_disabled == '1' then
        UNDEFINED;

    return;
```

Library pseudocode for aarch32/exceptions/traps/AArch32.SystemAccessTrap

```
// AArch32.SystemAccessTrap()
// ==========================
// Trapped  System register access.

AArch32.SystemAccessTrap(bits(5) mode, integer ec)
    (valid, target_el) = ELFromM32(mode);
    assert valid && HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    if target_el == EL2 then
        exception = AArch32.SystemAccessTrapSyndrome(ThisInstr(), ec);
        AArch32.TakeHypTrapException(exception);
    else
        AArch32.TakeUndefInstrException();
```

Library pseudocode for aarch32/exceptions/traps/AArch32.SystemAccessTrapSyndrome

```
// AArch32.SystemAccessTrapSyndrome()
// ==================================
// Returns the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS, VMSR instructior
// other than traps that are due to HCPTR or CPACR.

ExceptionRecord AArch32.SystemAccessTrapSyndrome(bits(32) instr, integer ec)
    ExceptionRecord exception;

    case ec of
        when 0x0    exception = ExceptionSyndrome(Exception_Uncategorized);
        when 0x3    exception = ExceptionSyndrome(Exception_CP15RTTrap);
        when 0x4    exception = ExceptionSyndrome(Exception_CP15RRTTrap);
        when 0x5    exception = ExceptionSyndrome(Exception_CP14RTTrap);
        when 0x6    exception = ExceptionSyndrome(Exception_CP14DTTrap);
        when 0x7    exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        when 0x8    exception = ExceptionSyndrome(Exception_FPIDTrap);
        when 0xC    exception = ExceptionSyndrome(Exception_CP14RRTTrap);
        otherwise   Unreachable();

    bits(20) iss = Zeros(20);

    if exception.exceptype == Exception_Uncategorized then
        return exception;
    elsif exception.exceptype IN {Exception_FPIDTrap, Exception_CP14RTTrap, Exception_CP15RTTrap} then
        // Trapped MRC/MCR, VMRS on FPSID
        iss<13:10> = instr<19:16>;        // CRn, Reg in case of VMRS
        iss<8:5>   = instr<15:12>;        // Rt
        iss<9>     = '0';                 // RES0

        if exception.exceptype != Exception_FPIDTrap then    // When trap is not for VMRS
            iss<19:17> = instr<7:5>;          // opc2
            iss<16:14> = instr<23:21>;        // opc1
            iss<4:1>   = instr<3:0>;          //CRm
        else //VMRS Access
            iss<19:17> = '000';               //opc2 - Hardcoded for VMRS
            iss<16:14> = '111';               //opc1 - Hardcoded for VMRS
            iss<4:1>   = '0000';              //CRm  - Hardcoded for VMRS
    elsif exception.exceptype IN {Exception_CP14RRTTrap, Exception_AdvSIMDFPAccessTrap, Exception_CP15RRT
        // Trapped MRRC/MCRR, VMRS/VMSR
        iss<19:16> = instr<7:4>;          // opc1
        iss<13:10> = instr<19:16>;        // Rt2
        iss<8:5> = instr<15:12>;          // Rt
        iss<4:1>   = instr<3:0>;         // CRm
    elsif exception.exceptype == Exception_CP14DTTrap then
        // Trapped LDC/STC
        iss<19:12> = instr<7:0>;         // imm8
        iss<4>     = instr<23>;          // U
        iss<2:1>   = instr<24,21>;       // P,W
        if instr<19:16> == '1111' then   // Rn==15, LDC(Literal addressing)/STC
            iss<8:5> = bits(4) UNKNOWN;
            iss<3>   = '1';
    iss<0> = instr<20>;                  // Direction

    exception.syndrome<24:20> = ConditionSyndrome();
    exception.syndrome<19:0>  = iss;

    return exception;
```

Library pseudocode for aarch32/exceptions/traps/AArch32.TakeHypTrapException

```
// AArch32.TakeHypTrapException()
// ==============================
// Exceptions routed to Hyp mode as a Hyp Trap exception.

AArch32.TakeHypTrapException(integer ec)
    exception = AArch32.SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch32.TakeHypTrapException(exception);

// AArch32.TakeHypTrapException()
// ==============================
// Exceptions routed to Hyp mode as a Hyp Trap exception.

AArch32.TakeHypTrapException(ExceptionRecord exception)
    assert HaveEL(EL2) && CurrentSecurityState() == SS_NonSecure && ELUsingAArch32(EL2);

    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x14;

    AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch32/exceptions/traps/AArch32.TakeMonitorTrapException

```
// AArch32.TakeMonitorTrapException()
// ==================================
// Exceptions routed to Monitor mode as a Monitor Trap exception.

AArch32.TakeMonitorTrapException()
    assert HaveEL(EL3) && ELUsingAArch32(EL3);

    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x04;
    lr_offset = if CurrentInstrSet() == InstrSet_A32 then 4 else 2;

    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/traps/AArch32.TakeUndefInstrException

```
// AArch32.TakeUndefInstrException()
// =================================

AArch32.TakeUndefInstrException()
    exception = ExceptionSyndrome(Exception_Uncategorized);
    AArch32.TakeUndefInstrException(exception);

// AArch32.TakeUndefInstrException()
// =================================

AArch32.TakeUndefInstrException(ExceptionRecord exception)

    route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';
    bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x04;
    lr_offset = if CurrentInstrSet() == InstrSet_A32 then 4 else 2;

    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    elsif route_to_hyp then
        AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.EnterMode(M32_Undef, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/traps/AArch32.UndefinedFault

```
// AArch32.UndefinedFault()
// ========================

AArch32.UndefinedFault()

    if AArch32.GeneralExceptionsToAArch64() then AArch64.UndefinedFault();
    AArch32.TakeUndefInstrException();
```

Library pseudocode for aarch32/functions/aborts/AArch32.DomainValid

```
// AArch32.DomainValid()
// =====================
// Returns TRUE if the Domain is valid for a Short-descriptor translation scheme.

boolean AArch32.DomainValid(Fault statuscode, integer level)
    assert statuscode != Fault_None;

    case statuscode of
        when Fault_Domain
            return TRUE;
        when Fault_Translation, Fault_AccessFlag, Fault_SyncExternalOnWalk, Fault_SyncParityOnWalk
            return level == 2;
        otherwise
            return FALSE;
```

Library pseudocode for aarch32/functions/aborts/AArch32.FaultStatusLD

```
// AArch32.FaultStatusLD()
// =======================
// Creates an exception fault status value for Abort and Watchpoint exceptions taken
// to Abort mode using AArch32 and Long-descriptor format.

bits(32) AArch32.FaultStatusLD(boolean d_side, FaultRecord fault)
    assert fault.statuscode != Fault_None;

    bits(32) fsr = Zeros(32);
    if HaveRASExt() && IsAsyncAbort(fault) then fsr<15:14> = fault.errortype;
    if d_side then
        if fault.acctype IN {AccType_DC, AccType_IC,
            AccType_AT, AccType_ATPAN} then
            fsr<13> = '1'; fsr<11> = '1';
        else
            fsr<11> = if fault.write then '1' else '0';
    if IsExternalAbort(fault) then fsr<12> = fault.extflag;
    fsr<9> = '1';
    fsr<5:0> = EncodeLDFSC(fault.statuscode, fault.level);

    return fsr;
```

Library pseudocode for aarch32/functions/aborts/AArch32.FaultStatusSD

```
// AArch32.FaultStatusSD()
// =======================
// Creates an exception fault status value for Abort and Watchpoint exceptions taken
// to Abort mode using AArch32 and Short-descriptor format.

bits(32) AArch32.FaultStatusSD(boolean d_side, FaultRecord fault)
    assert fault.statuscode != Fault_None;

    bits(32) fsr = Zeros(32);
    if HaveRASExt() && IsAsyncAbort(fault) then fsr<15:14> = fault.errortype;
    if d_side then
        if fault.acctype IN {AccType_DC, AccType_IC,
            AccType_AT, AccType_ATPAN} then
            fsr<13> = '1'; fsr<11> = '1';
        else
            fsr<11> = if fault.write then '1' else '0';
    if IsExternalAbort(fault) then fsr<12> = fault.extflag;
    fsr<9> = '0';
    fsr<10,3:0> = EncodeSDFSC(fault.statuscode, fault.level);
    if d_side then
        fsr<7:4> = fault.domain;               // Domain field (data fault only)

    return fsr;
```

Library pseudocode for aarch32/functions/aborts/AArch32.FaultSyndrome

```
// AArch32.FaultSyndrome()
// =======================
// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to
// AArch32 Hyp mode.

bits(25) AArch32.FaultSyndrome(boolean d_side, FaultRecord fault)
    assert fault.statuscode != Fault_None;

    bits(25) iss = Zeros(25);

    if HaveRASExt() && IsAsyncAbort(fault) then
        iss<11:10> = fault.errortype; // AET

    if d_side then
        if (IsSecondStage(fault) && !fault.s2fs1walk &&
            (!IsExternalSyncAbort(fault) ||
            (!HaveRASExt() && fault.acctype == AccType_TTW &&
            boolean IMPLEMENTATION_DEFINED "ISV on second stage translation table walk"))) then
            iss<24:14> = LSInstructionSyndrome();

        if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT, AccType_ATPAN} then
            iss<8> = '1';  iss<6> = '1';
        else
            iss<6> = if fault.write then '1' else '0';

    if IsExternalAbort(fault) then iss<9> = fault.extflag;
    iss<7> = if fault.s2fs1walk then '1' else '0';
    iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);

    return iss;
```

Library pseudocode for aarch32/functions/aborts/EncodeSDFSC

```
// EncodeSDFSC()
// =============
// Function that gives the Short-descriptor FSR code for different types of Fault

bits(5) EncodeSDFSC(Fault statuscode, integer level)

    bits(5) result;
    case statuscode of
        when Fault_AccessFlag
            assert level IN {1,2};
            result = if level == 1 then '00011' else '00110';
        when Fault_Alignment
            result = '00001';
        when Fault_Permission
            assert level IN {1,2};
            result = if level == 1 then '01101' else '01111';
        when Fault_Domain
            assert level IN {1,2};
            result = if level == 1 then '01001' else '01011';
        when Fault_Translation
            assert level IN {1,2};
            result = if level == 1 then '00101' else '00111';
        when Fault_SyncExternal
            result = '01000';
        when Fault_SyncExternalOnWalk
            assert level IN {1,2};
            result = if level == 1 then '01100' else '01110';
        when Fault_SyncParity
            result = '11001';
        when Fault_SyncParityOnWalk
            assert level IN {1,2};
            result = if level == 1 then '11100' else '11110';
        when Fault_AsyncParity
            result = '11000';
        when Fault_AsyncExternal
            result = '10110';
        when Fault_Debug
            result = '00010';
        when Fault_TLBConflict
            result = '10000';
        when Fault_Lockdown
            result = '10100';   // IMPLEMENTATION DEFINED
        when Fault_Exclusive
            result = '10101';   // IMPLEMENTATION DEFINED
        when Fault_ICacheMaint
            result = '00100';
        otherwise
            Unreachable();

    return result;
```

Library pseudocode for aarch32/functions/common/A32ExpandImm

```
// A32ExpandImm()
// ==============

bits(32) A32ExpandImm(bits(12) imm12)

    // PSTATE.C argument to following function call does not affect the imm32 result.
    (imm32, -) = A32ExpandImm_C(imm12, PSTATE.C);

    return imm32;
```

Library pseudocode for aarch32/functions/common/A32ExpandImm_C

```
// A32ExpandImm_C()
// ================

(bits(32), bit) A32ExpandImm_C(bits(12) imm12, bit carry_in)

    unrotated_value = ZeroExtend(imm12<7:0>, 32);
    (imm32, carry_out) = Shift_C(unrotated_value, SRType_ROR, 2*UInt(imm12<11:8>), carry_in);

    return (imm32, carry_out);
```

Library pseudocode for aarch32/functions/common/DecodeImmShift

```
// DecodeImmShift()
// ================

(SRType, integer) DecodeImmShift(bits(2) srtype, bits(5) imm5)

    SRType shift_t;
    integer shift_n;
    case srtype of
        when '00'
            shift_t = SRType_LSL;  shift_n = UInt(imm5);
        when '01'
            shift_t = SRType_LSR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '10'
            shift_t = SRType_ASR;  shift_n = if imm5 == '00000' then 32 else UInt(imm5);
        when '11'
            if imm5 == '00000' then
                shift_t = SRType_RRX;  shift_n = 1;
            else
                shift_t = SRType_ROR;  shift_n = UInt(imm5);

    return (shift_t, shift_n);
```

Library pseudocode for aarch32/functions/common/DecodeRegShift

```
// DecodeRegShift()
// ================

SRType DecodeRegShift(bits(2) srtype)
    SRType shift_t;
    case srtype of
        when '00'  shift_t = SRType_LSL;
        when '01'  shift_t = SRType_LSR;
        when '10'  shift_t = SRType_ASR;
        when '11'  shift_t = SRType_ROR;
    return shift_t;
```

Library pseudocode for aarch32/functions/common/RRX

```
// RRX()
// =====

bits(N) RRX(bits(N) x, bit carry_in)
    (result, -) = RRX_C(x, carry_in);
    return result;
```

Library pseudocode for aarch32/functions/common/RRX_C

```
// RRX_C()
// =======

(bits(N), bit) RRX_C(bits(N) x, bit carry_in)
    result = carry_in : x<N-1:1>;
    carry_out = x<0>;
    return (result, carry_out);
```

Library pseudocode for aarch32/functions/common/SRType

```
enumeration SRType {SRType_LSL, SRType_LSR, SRType_ASR, SRType_ROR, SRType_RRX};
```

Library pseudocode for aarch32/functions/common/Shift

```
// Shift()
// =======

bits(N) Shift(bits(N) value, SRType srtype, integer amount, bit carry_in)
    (result, -) = Shift_C(value, srtype, amount, carry_in);
    return result;
```

Library pseudocode for aarch32/functions/common/Shift_C

```
// Shift_C()
// =========

(bits(N), bit) Shift_C(bits(N) value, SRType srtype, integer amount, bit carry_in)
    assert !(srtype == SRType_RRX && amount != 1);

    bits(N) result;
    bit carry_out;
    if amount == 0 then
        (result, carry_out) = (value, carry_in);
    else
        case srtype of
            when SRType_LSL
                (result, carry_out) = LSL_C(value, amount);
            when SRType_LSR
                (result, carry_out) = LSR_C(value, amount);
            when SRType_ASR
                (result, carry_out) = ASR_C(value, amount);
            when SRType_ROR
                (result, carry_out) = ROR_C(value, amount);
            when SRType_RRX
                (result, carry_out) = RRX_C(value, carry_in);

    return (result, carry_out);
```

Library pseudocode for aarch32/functions/common/T32ExpandImm

```
// T32ExpandImm()
// ==============

bits(32) T32ExpandImm(bits(12) imm12)

    // PSTATE.C argument to following function call does not affect the imm32 result.
    (imm32, -) = T32ExpandImm_C(imm12, PSTATE.C);

    return imm32;
```

Library pseudocode for aarch32/functions/common/T32ExpandImm_C

```
// T32ExpandImm_C()
// ================

(bits(32), bit) T32ExpandImm_C(bits(12) imm12, bit carry_in)
    bits(32) imm32;
    bit carry_out;
    if imm12<11:10> == '00' then
        case imm12<9:8> of
            when '00'
                imm32 = ZeroExtend(imm12<7:0>, 32);
            when '01'
                imm32 = '00000000' : imm12<7:0> : '00000000' : imm12<7:0>;
            when '10'
                imm32 = imm12<7:0> : '00000000' : imm12<7:0> : '00000000';
            when '11'
                imm32 = imm12<7:0> : imm12<7:0> : imm12<7:0> : imm12<7:0>;
        carry_out = carry_in;
    else
        unrotated_value = ZeroExtend('1':imm12<6:0>, 32);
        (imm32, carry_out) = ROR_C(unrotated_value, UInt(imm12<11:7>));

    return (imm32, carry_out);
```

Library pseudocode for aarch32/functions/common/VBitOps

```
enumeration VBitOps {VBitOps_VBIF, VBitOps_VBIT, VBitOps_VBSL};
```

Library pseudocode for aarch32/functions/common/VCGEType

```
enumeration VCGEType {VCGEType_signed, VCGEType_unsigned, VCGEType_fp};
```

Library pseudocode for aarch32/functions/common/VCGTtype

```
enumeration VCGTtype {VCGTtype_signed, VCGTtype_unsigned, VCGTtype_fp};
```

Library pseudocode for aarch32/functions/common/VFPNegMul

```
enumeration VFPNegMul {VFPNegMul_VNMLA, VFPNegMul_VNMLS, VFPNegMul_VNMUL};
```

Library pseudocode for aarch32/functions/coproc/AArch32.CheckCP15InstrCoarseTraps

```
// AArch32.CheckCP15InstrCoarseTraps()
// ===================================
// Check for coarse-grained  traps to System registers in the
// coproc=0b1111 encoding space by HSTR and HCR.

AArch32.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)
    if PSTATE.EL == EL0 && (!ELUsingAArch32(EL1) ||
            (EL2Enabled() && !ELUsingAArch32(EL2))) then
        AArch64.CheckCP15InstrCoarseTraps(CRn, nreg, CRm);

    trapped_encoding = ((CRn == 9  && CRm IN {0,1,2,    5,6,7,8   }) ||
                        (CRn == 10 && CRm IN {0,1,    4,      8   }) ||
                        (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}));

    // Check for coarse-grained Hyp traps
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        major = if nreg == 1 then CRn else CRm;
        // Check for MCR, MRC, MCRR, and MRRC disabled by HSTR<CRn/CRm>
        // and MRC and MCR disabled by HCR.TIDCP.
        if ((!(major IN {4,14}) && HSTR<major> == '1') ||
                (HCR.TIDCP == '1' && nreg == 1 && trapped_encoding)) then
            if (PSTATE.EL == EL0 &&
                    boolean IMPLEMENTATION_DEFINED "UNDEF unallocated CP15 access at EL0") then
                UNDEFINED;
            if ELUsingAArch32(EL2) then
                AArch32.SystemAccessTrap(M32_Hyp, 0x3);
            else
                AArch64.AArch32SystemAccessTrap(EL2, 0x3);
```

Library pseudocode for aarch32/functions/exclusive/AArch32.ExclusiveMonitorsPass

```
// AArch32.ExclusiveMonitorsPass()
// ===============================
// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

boolean AArch32.ExclusiveMonitorsPass(bits(32) address, integer size)

    // It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
    // before or after the check on the local Exclusives monitor. As a result a failure
    // of the local monitor can occur on some implementations even if the memory
    // access would give an memory abort.

    acctype = AccType_ATOMIC;
    iswrite = TRUE;

    aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);

    passed = AArch32.IsExclusiveVA(address, ProcessorID(), size);
    if !passed then
        return FALSE;

    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
    ClearExclusiveLocal(ProcessorID());

    if passed then
        if memaddrdesc.memattrs.shareability != Shareability_NSH then
            passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    return passed;
```

Library pseudocode for aarch32/functions/exclusive/AArch32.IsExclusiveVA

```
// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address only.
boolean AArch32.IsExclusiveVA(bits(32) address, integer processorid, integer size);
```

Library pseudocode for aarch32/functions/exclusive/AArch32.MarkExclusiveVA

```
// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.
AArch32.MarkExclusiveVA(bits(32) address, integer processorid, integer size);
```

Library pseudocode for aarch32/functions/exclusive/AArch32.SetExclusiveMonitors

```
// AArch32.SetExclusiveMonitors()
// ==============================
// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

AArch32.SetExclusiveMonitors(bits(32) address, integer size)
    acctype = AccType_ATOMIC;
    iswrite = FALSE;

    aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);

    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;

    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    AArch32.MarkExclusiveVA(address, ProcessorID(), size);
```

Library pseudocode for aarch32/functions/float/CheckAdvSIMDEnabled

```
// CheckAdvSIMDEnabled()
// =====================

CheckAdvSIMDEnabled()

    fpexc_check = TRUE;
    advsimd = TRUE;

    AArch32.CheckAdvSIMDOrFPEnabled(fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEnabled() occurs only if Advanced SIMD access is permitted

    // Make temporary copy of D registers
    // _Dclone[] is used as input data for instruction pseudocode
    for i = 0 to 31
        _Dclone[i] = D[i];

    return;
```

Library pseudocode for aarch32/functions/float/CheckAdvSIMDOrVFPEnabled

```
// CheckAdvSIMDOrVFPEnabled()
// ==========================

CheckAdvSIMDOrVFPEnabled(boolean include_fpexc_check, boolean advsimd)
    AArch32.CheckAdvSIMDOrFPEnabled(include_fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEnabled() occurs only if VFP access is permitted
    return;
```

Library pseudocode for aarch32/functions/float/CheckCryptoEnabled32

```
// CheckCryptoEnabled32()
// ======================

CheckCryptoEnabled32()
    CheckAdvSIMDEnabled();
    // Return from CheckAdvSIMDEnabled() occurs only if access is permitted
    return;
```

Library pseudocode for aarch32/functions/float/CheckVFPEnabled

```
// CheckVFPEnabled()
// =================

CheckVFPEnabled(boolean include_fpexc_check)
    advsimd = FALSE;
    AArch32.CheckAdvSIMDOrFPEnabled(include_fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEnabled() occurs only if VFP access is permitted
    return;
```

Library pseudocode for aarch32/functions/float/FPHalvedSub

```
// FPHalvedSub()
// =============

bits(N) FPHalvedSub(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN(fpcr, N);
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0', N);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1', N);
        elsif zero1 && zero2 && sign1 != sign2 then
            result = FPZero(sign1, N);
        else
            result_value = (value1 - value2) / 2.0;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, N);
    return result;
```

Library pseudocode for aarch32/functions/float/FPRSqrtStep

```
// FPRSqrtStep()
// =============

bits(N) FPRSqrtStep(bits(N) op1, bits(N) op2)
    assert N IN {16,32};
    FPCRType fpcr = StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        bits(N) product;
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero('0', N);
        else
            product = FPMul(op1, op2, fpcr);
        bits(N) three = FPThree('0', N);
        result = FPHalvedSub(three, product, fpcr);
    return result;
```

Library pseudocode for aarch32/functions/float/FPRecipStep

```
// FPRecipStep()
// =============

bits(N) FPRecipStep(bits(N) op1, bits(N) op2)
    assert N IN {16,32};
    FPCRType fpcr = StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        bits(N) product;
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero('0', N);
        else
            product = FPMul(op1, op2, fpcr);
        bits(N) two = FPTwo('0', N);
        result = FPSub(two, product, fpcr);
    return result;
```

Library pseudocode for aarch32/functions/float/StandardFPSCRValue

```
// StandardFPSCRValue()
// ====================

FPCRType StandardFPSCRValue()
    bits(32) value = '00000' : FPSCR.AHP : '110000' : FPSCR.FZ16 : '0000000000000000000';
    return ZeroExtend(value, 64);
```

Library pseudocode for aarch32/functions/memory/AArch32.CheckAlignment

```
// AArch32.CheckAlignment()
// ========================

boolean AArch32.CheckAlignment(bits(32) address, integer alignment, AccType acctype,
                               boolean iswrite)

    bit A;
    if PSTATE.EL == EL0 && !ELUsingAArch32(S1TranslationRegime()) then
        A = SCTLR[].A; //use AArch64 register, when higher Exception level is using AArch64
    elsif PSTATE.EL == EL2 then
        A = HSCTLR.A;
    else
        A = SCTLR.A;
    aligned = (address == Align(address, alignment));
    atomic  = acctype IN { AccType_ATOMIC, AccType_ATOMICRW, AccType_ORDEREDATOMIC,
                           AccType_ORDEREDATOMICRW, AccType_ATOMICLS64, AccType_A32LSMD};
    ordered = acctype IN { AccType_ORDERED, AccType_ORDEREDRW, AccType_LIMITEDORDERED,
                           AccType_ORDEREDATOMIC, AccType_ORDEREDATOMICRW };

    vector  = acctype == AccType_VEC;
    // AccType_VEC is used for SIMD element alignment checks only
    check = (atomic || ordered || vector || A == '1');

    if check && !aligned then
        secondstage = FALSE;
        AArch32.Abort(address, AlignmentFault(acctype, iswrite, secondstage));

    return aligned;
```

Library pseudocode for aarch32/functions/memory/AArch32.MemSingle

```
// AArch32.MemSingle[] - non-assignment (read) form
// ================================================
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean aligned]
    boolean ispair = FALSE;
    return AArch32.MemSingle[address, size, acctype, aligned, ispair];

// AArch32.MemSingle[] - non-assignment (read) form
// ================================================
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean aligned, boolean
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    bits(size*8) value;
    iswrite = FALSE;

    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);

    PhysMemRetStatus memstatus;
    (memstatus, value) = PhysMemRead(memaddrdesc, size, accdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, size, accdesc);
    return value;

// AArch32.MemSingle[] - assignment (write) form
// =============================================

AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean aligned] = bits(size*8) value
    boolean ispair = FALSE;
    AArch32.MemSingle[address, size, acctype, aligned, ispair] = value;
    return;

// AArch32.MemSingle[] - assignment (write) form
// =============================================
// Perform an atomic, little-endian write of 'size' bytes.

AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean aligned, boolean ispair] = bit
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    iswrite = TRUE;

    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);

    PhysMemRetStatus memstatus;
    memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);
    return;
```

Library pseudocode for aarch32/functions/memory/Hint_PreloadData

```
Hint_PreloadData(bits(32) address);
```

Library pseudocode for aarch32/functions/memory/Hint_PreloadDataForWrite

```
Hint_PreloadDataForWrite(bits(32) address);
```

Library pseudocode for aarch32/functions/memory/Hint_PreloadInstr

```
Hint_PreloadInstr(bits(32) address);
```

Library pseudocode for aarch32/functions/memory/MemA

```
// MemA[] - non-assignment form
// ============================

bits(8*size) MemA[bits(32) address, integer size]
    acctype = AccType_ATOMIC;
    return Mem_with_type[address, size, acctype];

// MemA[] - assignment form
// ========================

MemA[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType_ATOMIC;
    Mem_with_type[address, size, acctype] = value;
    return;
```

Library pseudocode for aarch32/functions/memory/MemO

```
// MemO[] - non-assignment form
// ============================

bits(8*size) MemO[bits(32) address, integer size]
    acctype = AccType_ORDERED;
    return Mem_with_type[address, size, acctype];

// MemO[] - assignment form
// ========================

MemO[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType_ORDERED;
    Mem_with_type[address, size, acctype] = value;
    return;
```

Library pseudocode for aarch32/functions/memory/MemS

```
// MemS[] - non-assignment form
// ============================
// Memory accessor for streaming load multiple instructions

bits(8*size) MemS[bits(32) address, integer size]
    acctype = AccType_A32LSMD;
    return Mem_with_type[address, size, acctype];

// MemS[] - assignment form
// ========================
// Memory accessor for streaming store multiple instructions

MemS[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType_A32LSMD;
    Mem_with_type[address, size, acctype] = value;
    return;
```

Library pseudocode for aarch32/functions/memory/MemU

```
// MemU[] - non-assignment form
// ============================

bits(8*size) MemU[bits(32) address, integer size]
    acctype = AccType_NORMAL;
    return Mem_with_type[address, size, acctype];

// MemU[] - assignment form
// ========================

MemU[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType_NORMAL;
    Mem_with_type[address, size, acctype] = value;
    return;
```

Library pseudocode for aarch32/functions/memory/MemU_unpriv

```
// MemU_unpriv[] - non-assignment form
// ===================================

bits(8*size) MemU_unpriv[bits(32) address, integer size]
    acctype = AccType_UNPRIV;
    return Mem_with_type[address, size, acctype];

// MemU_unpriv[] - assignment form
// ===============================

MemU_unpriv[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType_UNPRIV;
    Mem_with_type[address, size, acctype] = value;
    return;
```

Library pseudocode for aarch32/functions/memory/Mem_with_type

```
// Mem_with_type[] - non-assignment (read) form
// ============================================
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch32.MemSingle directly.

bits(size*8) Mem_with_type[bits(32) address, integer size, AccType acctype]
    boolean ispair = FALSE;
    return Mem_with_type[address, size, acctype, ispair];

bits(size*8) Mem_with_type[bits(32) address, integer size, AccType acctype, boolean ispair]
    assert size IN {1, 2, 4, 8, 16};
    constant halfsize = size DIV 2;
    bits(size * 8) value;
    boolean iswrite = FALSE;
    boolean aligned;
    if ispair then
        // check alignment on size of element accessed, not overall access size
        aligned = AArch32.CheckAlignment(address, halfsize, acctype, iswrite);
    else
        aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);

    if !aligned then

        assert size > 1;
        value<7:0> = AArch32.MemSingle[address, 1, acctype, aligned];

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
        assert c IN {Constraint_FAULT, Constraint_NONE};
        if c == Constraint_NONE then aligned = TRUE;
        for i = 1 to size-1
            value<8*i+7:8*i> = AArch32.MemSingle[address+i, 1, acctype, aligned];

    else
        value = AArch32.MemSingle[address, size, acctype, aligned, ispair];

    if BigEndian(acctype) then
        value = BigEndianReverse(value);

    return value;

// Mem_with_type[] - assignment (write) form
// =========================================
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem_with_type[bits(32) address, integer size, AccType acctype] = bits(size*8) value_in
    boolean ispair = FALSE;
    Mem_with_type[address, size, acctype, ispair] = value_in;

Mem_with_type[bits(32) address, integer size, AccType acctype, boolean ispair] = bits(size*8) value_in
    boolean iswrite = TRUE;
    constant halfsize = size DIV 2;
    bits(size*8) value = value_in;
    boolean aligned;
    if BigEndian(acctype) then
        value = BigEndianReverse(value);

    if ispair then
        // check alignment on size of element accessed, not overall access size
        aligned = AArch32.CheckAlignment(address, halfsize, acctype, iswrite);
    else
        aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);

    if !aligned then
        assert size > 1;
        AArch32.MemSingle[address, 1, acctype, aligned] = value<7:0>;

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
```

```
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
        assert c IN {Constraint_FAULT, Constraint_NONE};
        if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            AArch32.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
    else
        AArch32.MemSingle[address, size, acctype, aligned, ispair] = value;
    return;
```

Library pseudocode for aarch32/functions/ras/AArch32.ESBOperation

```
// AArch32.ESBOperation()
// ======================
// Perform the AArch32 ESB operation for ESB executed in AArch32 state

AArch32.ESBOperation()

    // Check if routed to AArch64 state
    route_to_aarch64 = PSTATE.EL == EL0 && !ELUsingAArch32(EL1);
    if !route_to_aarch64 && EL2Enabled() && !ELUsingAArch32(EL2) then
        route_to_aarch64 = HCR_EL2.TGE == '1' || HCR_EL2.AMO == '1';
    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.EA == '1';

    if route_to_aarch64 then
        AArch64.ESBOperation();
        return;

    route_to_monitor = HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.EA == '1';
    route_to_hyp = PSTATE.EL IN {EL0, EL1} && EL2Enabled() && (HCR.TGE == '1' || HCR.AMO == '1');

    bits(5) target;
    if route_to_monitor then
        target = M32_Monitor;
    elsif route_to_hyp || PSTATE.M == M32_Hyp then
        target = M32_Hyp;
    else
        target = M32_Abort;

    boolean mask_active;
    if CurrentSecurityState() == SS_Secure then
        mask_active = TRUE;
    elsif target == M32_Monitor then
        mask_active = SCR.AW == '1' && (!HaveEL(EL2) || (HCR.TGE == '0' && HCR.AMO == '0'));
    else
        mask_active = target == M32_Abort || PSTATE.M == M32_Hyp;

    mask_set = PSTATE.A == '1';
    (-, el)  = ELFromM32(target);
    intdis   = Halted() || ExternalDebugInterruptsDisabled(el);
    masked   = intdis || (mask_active && mask_set);

    // Check for a masked Physical SError pending that can be synchronized
    // by an Error synchronization event.
    if masked && IsSynchronizablePhysicalSErrorPending() then
        syndrome32 = AArch32.PhysicalSErrorSyndrome();
        DISR = AArch32.ReportDeferredSError(syndrome32.AET, syndrome32.ExT);
        ClearPendingPhysicalSError();

    return;
```

Library pseudocode for aarch32/functions/ras/AArch32.PhysicalSErrorSyndrome

```
// Return the SError syndrome
AArch32.SErrorSyndrome AArch32.PhysicalSErrorSyndrome();
```

Library pseudocode for aarch32/functions/ras/AArch32.ReportDeferredSError

```
// AArch32.ReportDeferredSError()
// ==============================
// Return deferred SError syndrome

bits(32) AArch32.ReportDeferredSError(bits(2) AET, bit ExT)
    bits(32) target;
    target<31> = '1';                           // A
    syndrome = Zeros(16);
    if PSTATE.EL == EL2 then
        syndrome<11:10> = AET;                  // AET
        syndrome<9>     = ExT;                  // EA
        syndrome<5:0>   = '010001';             // DFSC
    else
        syndrome<15:14> = AET;                  // AET
        syndrome<12>    = ExT;                  // ExT
        syndrome<9>     = TTBCR.EAE;            // LPAE
        if TTBCR.EAE == '1' then                // Long-descriptor format
            syndrome<5:0>    = '010001';        // STATUS
        else                                    // Short-descriptor format
            syndrome<10,3:0> = '10110';         // FS
    if HaveAArch64() then
        target<24:0> = ZeroExtend(syndrome, 25);// Any RES0 fields must be set to zero
    else
        target<15:0> = syndrome;
    return target;
```

Library pseudocode for aarch32/functions/ras/AArch32.SErrorSyndrome

```
type AArch32.SErrorSyndrome is (
    bits(2) AET,
    bit ExT
)
```

Library pseudocode for aarch32/functions/ras/AArch32.vESBOperation

```
// AArch32.vESBOperation()
// =======================
// Perform the ESB operation for virtual SError interrupts executed in AArch32 state

AArch32.vESBOperation()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    // Check for EL2 using AArch64 state
    if !ELUsingAArch32(EL2) then
        AArch64.vESBOperation();
        return;

    // If physical SError interrupts are routed to Hyp mode, and TGE is not set, then a
    // virtual SError interrupt might be pending
    vSEI_enabled = HCR.TGE == '0' && HCR.AMO == '1';
    vSEI_pending = vSEI_enabled && HCR.VA == '1';
    vintdis      = Halted() || ExternalDebugInterruptsDisabled(EL1);
    vmasked      = vintdis || PSTATE.A == '1';

    // Check for a masked virtual SError pending
    if vSEI_pending && vmasked then
        VDISR = AArch32.ReportDeferredSError(VDFSR<15:14>, VDFSR<12>);
        HCR.VA = '0';                       // Clear pending virtual SError

    return;
```

Library pseudocode for aarch32/functions/registers/AArch32.ResetGeneralRegisters

```
// AArch32.ResetGeneralRegisters()
// ===============================

AArch32.ResetGeneralRegisters()

    for i = 0 to 7
        R[i] = bits(32) UNKNOWN;
    for i = 8 to 12
        Rmode[i, M32_User] = bits(32) UNKNOWN;
        Rmode[i, M32_FIQ] = bits(32) UNKNOWN;
    if HaveEL(EL2) then Rmode[13, M32_Hyp] = bits(32) UNKNOWN;   // No R14_hyp
    for i = 13 to 14
        Rmode[i, M32_User] = bits(32) UNKNOWN;
        Rmode[i, M32_FIQ] = bits(32) UNKNOWN;
        Rmode[i, M32_IRQ] = bits(32) UNKNOWN;
        Rmode[i, M32_Svc] = bits(32) UNKNOWN;
        Rmode[i, M32_Abort] = bits(32) UNKNOWN;
        Rmode[i, M32_Undef] = bits(32) UNKNOWN;
        if HaveEL(EL3) then Rmode[i, M32_Monitor] = bits(32) UNKNOWN;

    return;
```

Library pseudocode for aarch32/functions/registers/AArch32.ResetSIMDFPRegisters

```
// AArch32.ResetSIMDFPRegisters()
// ==============================

AArch32.ResetSIMDFPRegisters()

    for i = 0 to 15
        Q[i] = bits(128) UNKNOWN;

    return;
```

Library pseudocode for aarch32/functions/registers/AArch32.ResetSpecialRegisters

```
// AArch32.ResetSpecialRegisters()
// ===============================

AArch32.ResetSpecialRegisters()

    // AArch32 special registers
    SPSR_fiq<31:0> = bits(32) UNKNOWN;
    SPSR_irq<31:0> = bits(32) UNKNOWN;
    SPSR_svc<31:0> = bits(32) UNKNOWN;
    SPSR_abt<31:0> = bits(32) UNKNOWN;
    SPSR_und<31:0> = bits(32) UNKNOWN;
    if HaveEL(EL2) then
        SPSR_hyp = bits(32) UNKNOWN;
        ELR_hyp = bits(32) UNKNOWN;
    if HaveEL(EL3) then
        SPSR_mon = bits(32) UNKNOWN;

    // External debug special registers
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;

    return;
```

Library pseudocode for aarch32/functions/registers/AArch32.ResetSystemRegisters

```
AArch32.ResetSystemRegisters(boolean cold_reset);
```

Library pseudocode for aarch32/functions/registers/ALUExceptionReturn

```
// ALUExceptionReturn()
// ====================

ALUExceptionReturn(bits(32) address)
    if PSTATE.EL == EL2 then
        UNDEFINED;
    elsif PSTATE.M IN {M32_User,M32_System} then
        Constraint c = ConstrainUnpredictable(Unpredictable_ALUEXCEPTIONRETURN);
        assert c IN {Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNDEF
                UNDEFINED;
            when Constraint_NOP
                EndOfInstruction();
    else
        AArch32.ExceptionReturn(address, SPSR[]);
```

Library pseudocode for aarch32/functions/registers/ALUWritePC

```
// ALUWritePC()
// ============

ALUWritePC(bits(32) address)
    if CurrentInstrSet() == InstrSet_A32 then
        BXWritePC(address, BranchType_INDIR);
    else
        BranchWritePC(address, BranchType_INDIR);
```

Library pseudocode for aarch32/functions/registers/BXWritePC

```
// BXWritePC()
// ===========

BXWritePC(bits(32) address_in, BranchType branch_type)
    bits(32) address = address_in;
    if address<0> == '1' then
        SelectInstrSet(InstrSet_T32);
        address<0> = '0';
    else
        SelectInstrSet(InstrSet_A32);
        // For branches to an unaligned PC counter in A32 state, the processor takes the branch
        // and does one of:
        // * Forces the address to be aligned
        // * Leaves the PC unaligned, meaning the target generates a PC Alignment fault.
        if address<1> == '1' && ConstrainUnpredictableBool(Unpredictable_A32FORCEALIGNPC) then
            address<1> = '0';
    boolean branch_conditional = !(AArch32.CurrentCond() IN {'111x'});
    BranchTo(address, branch_type, branch_conditional);
```

Library pseudocode for aarch32/functions/registers/BranchWritePC

```
// BranchWritePC()
// ===============

BranchWritePC(bits(32) address_in, BranchType branch_type)
    bits(32) address = address_in;
    if CurrentInstrSet() == InstrSet_A32 then
        address<1:0> = '00';
    else
        address<0> = '0';
    boolean branch_conditional = !(AArch32.CurrentCond() IN {'111x'});
    BranchTo(address, branch_type, branch_conditional);
```

Library pseudocode for aarch32/functions/registers/CBWritePC

```
// CBWritePC()
// ===========
// Takes a branch from a CBNZ/CBZ instruction.

CBWritePC(bits(32) address_in)
    bits(32) address = address_in;
    assert CurrentInstrSet() == InstrSet_T32;
    address<0> = '0';
    boolean branch_conditional = TRUE;
    BranchTo(address, BranchType_DIR, branch_conditional);
```

Library pseudocode for aarch32/functions/registers/D

```
// D[] - non-assignment form
// =========================

bits(64) D[integer n]
    assert n >= 0 && n <= 31;
    base = (n MOD 2) * 64;
    bits(128) vreg = V[n DIV 2, 128];
    return vreg<base+63:base>;

// D[] - assignment form
// =====================

D[integer n] = bits(64) value
    assert n >= 0 && n <= 31;
    base = (n MOD 2) * 64;
    bits(128) vreg = V[n DIV 2, 128];
    vreg<base+63:base> = value;
    V[n DIV 2, 128] = vreg;
    return;
```

Library pseudocode for aarch32/functions/registers/Din

```
// Din[] - non-assignment form
// ===========================

bits(64) Din[integer n]
    assert n >= 0 && n <= 31;
    return _Dclone[n];
```

Library pseudocode for aarch32/functions/registers/LR

```
// LR - assignment form
// ====================

LR = bits(32) value
    R[14] = value;
    return;

// LR - non-assignment form
// ========================

bits(32) LR
    return R[14];
```

Library pseudocode for aarch32/functions/registers/LoadWritePC

```
// LoadWritePC()
// =============

LoadWritePC(bits(32) address)
    BXWritePC(address, BranchType_INDIR);
```

Library pseudocode for aarch32/functions/registers/LookUpRIndex

```
// LookUpRIndex()
// ==============

integer LookUpRIndex(integer n, bits(5) mode)
    assert n >= 0 && n <= 14;

    integer result;
    case n of  // Select  index by mode:     usr fiq irq svc abt und hyp
        when 8     result = RBankSelect(mode,  8, 24,  8,  8,  8,  8,  8);
        when 9     result = RBankSelect(mode,  9, 25,  9,  9,  9,  9,  9);
        when 10    result = RBankSelect(mode, 10, 26, 10, 10, 10, 10, 10);
        when 11    result = RBankSelect(mode, 11, 27, 11, 11, 11, 11, 11);
        when 12    result = RBankSelect(mode, 12, 28, 12, 12, 12, 12, 12);
        when 13    result = RBankSelect(mode, 13, 29, 17, 19, 21, 23, 15);
        when 14    result = RBankSelect(mode, 14, 30, 16, 18, 20, 22, 14);
        otherwise  result = n;

    return result;
```

Library pseudocode for aarch32/functions/registers/Monitor_mode_registers

```
bits(32) SP_mon;
bits(32) LR_mon;
```

Library pseudocode for aarch32/functions/registers/PC

```
// PC - non-assignment form
// ========================

bits(32) PC
    return R[15];               // This includes the offset from AArch32 state
```

Library pseudocode for aarch32/functions/registers/PCStoreValue

```
// PCStoreValue()
// ==============

bits(32) PCStoreValue()
    // This function returns the PC value. On architecture versions before Armv7, it
    // is permitted to instead return PC+4, provided it does so consistently. It is
    // used only to describe A32 instructions, so it returns the address of the current
    // instruction plus 8 (normally) or 12 (when the alternative is permitted).
    return PC;
```

Library pseudocode for aarch32/functions/registers/Q

```
// Q[] - non-assignment form
// =========================

bits(128) Q[integer n]
    assert n >= 0 && n <= 15;
    return V[n, 128];

// Q[] - assignment form
// =====================

Q[integer n] = bits(128) value
    assert n >= 0 && n <= 15;
    V[n, 128] = value;
    return;
```

Library pseudocode for aarch32/functions/registers/Qin

```
// Qin[] - non-assignment form
// ===========================

bits(128) Qin[integer n]
    assert n >= 0 && n <= 15;
    return Din[2*n+1]:Din[2*n];
```

Library pseudocode for aarch32/functions/registers/R

```
// R[] - assignment form
// =====================

R[integer n] = bits(32) value
    Rmode[n, PSTATE.M] = value;
    return;

// R[] - non-assignment form
// =========================

bits(32) R[integer n]
    if n == 15 then
        offset = (if CurrentInstrSet() == InstrSet_A32 then 8 else 4);
        return _PC<31:0> + offset;
    else
        return Rmode[n, PSTATE.M];
```

Library pseudocode for aarch32/functions/registers/RBankSelect

```
// RBankSelect()
// =============

integer RBankSelect(bits(5) mode, integer usr, integer fiq, integer irq,
                    integer svc, integer abt, integer und, integer hyp)

    integer result;
    case mode of
        when M32_User    result = usr;  // User mode
        when M32_FIQ     result = fiq;  // FIQ mode
        when M32_IRQ     result = irq;  // IRQ mode
        when M32_Svc     result = svc;  // Supervisor mode
        when M32_Abort   result = abt;  // Abort mode
        when M32_Hyp     result = hyp;  // Hyp mode
        when M32_Undef   result = und;  // Undefined mode
        when M32_System  result = usr;  // System mode uses User mode registers
        otherwise        Unreachable(); // Monitor mode

    return result;
```

Library pseudocode for aarch32/functions/registers/Rmode

```
// Rmode[] - non-assignment form
// =============================

bits(32) Rmode[integer n, bits(5) mode]
    assert n >= 0 && n <= 14;

    // Check for attempted use of Monitor mode in Non-secure state.
    if CurrentSecurityState() != SS_Secure then assert mode != M32_Monitor;
    assert !BadMode(mode);

    if mode == M32_Monitor then
        if n == 13 then return SP_mon;
        elsif n == 14 then return LR_mon;
        else return _R[n]<31:0>;
    else
        return _R[LookUpRIndex(n, mode)]<31:0>;

// Rmode[] - assignment form
// =========================

Rmode[integer n, bits(5) mode] = bits(32) value
    assert n >= 0 && n <= 14;

    // Check for attempted use of Monitor mode in Non-secure state.
    if CurrentSecurityState() != SS_Secure then assert mode != M32_Monitor;
    assert !BadMode(mode);

    if mode == M32_Monitor then
        if n == 13 then SP_mon = value;
        elsif n == 14 then LR_mon = value;
        else _R[n]<31:0> = value;
    else
        // It is CONSTRAINED UNPREDICTABLE whether the upper 32 bits of the X
        // register are unchanged or set to zero. This is also tested for on
        // exception entry, as this applies to all AArch32 registers.
        if HaveAArch64() && ConstrainUnpredictableBool(Unpredictable_ZEROUPPER) then
            _R[LookUpRIndex(n, mode)] = ZeroExtend(value, 64);
        else
            _R[LookUpRIndex(n, mode)]<31:0> = value;

    return;
```

Library pseudocode for aarch32/functions/registers/S

```
// S[] - non-assignment form
// =========================

bits(32) S[integer n]
    assert n >= 0 && n <= 31;
    base = (n MOD 4) * 32;
    bits(128) vreg = V[n DIV 4, 128];
    return vreg<base+31:base>;

// S[] - assignment form
// =====================

S[integer n] = bits(32) value
    assert n >= 0 && n <= 31;
    base = (n MOD 4) * 32;
    bits(128) vreg = V[n DIV 4, 128];
    vreg<base+31:base> = value;
    V[n DIV 4, 128] = vreg;
    return;
```

Library pseudocode for aarch32/functions/registers/_Dclone

```
array bits(64) _Dclone[0..31];
```

Library pseudocode for aarch32/functions/system/AArch32.ExceptionReturn

```
// AArch32.ExceptionReturn()
// =========================

AArch32.ExceptionReturn(bits(32) new_pc_in, bits(32) spsr)
    bits(32) new_pc = new_pc_in;
    SynchronizeContext();
    // Attempts to change to an illegal mode or state will invoke the Illegal Execution state
    // mechanism
    SetPSTATEFromPSR(spsr);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();

    if PSTATE.IL == '1' then
        // If the exception return is illegal, PC[1:0] are UNKNOWN
        new_pc<1:0> = bits(2) UNKNOWN;
    else
        // LR[1:0] or LR[0] are treated as being 0, depending on the target instruction set state
        if PSTATE.T == '1' then
            new_pc<0> = '0';                // T32
        else
            new_pc<1:0> = '00';             // A32

    boolean branch_conditional = !(AArch32.CurrentCond() IN {'111x'});
    BranchTo(new_pc, BranchType_ERET, branch_conditional);

    CheckExceptionCatch(FALSE);             // Check for debug event on exception return
```

Library pseudocode for aarch32/functions/system/AArch32.ExecutingCP10or11Instr

```
// AArch32.ExecutingCP10or11Instr()
// ================================

boolean AArch32.ExecutingCP10or11Instr()
    instr =  ThisInstr();
    instr_set = CurrentInstrSet();
    assert instr_set IN {InstrSet_A32, InstrSet_T32};

    if instr_set == InstrSet_A32 then
        return ((instr<27:24> == '1110' || instr<27:25> == '110') && instr<11:8> IN {'101x'});
    else // InstrSet_T32
        return (instr<31:28> IN {'111x'} && (instr<27:24> == '1110' || instr<27:25> == '110') && instr<11
```

Library pseudocode for aarch32/functions/system/AArch32.ITAdvance

```
// AArch32.ITAdvance()
// ===================

AArch32.ITAdvance()
    if PSTATE.IT<2:0> == '000' then
        PSTATE.IT = '00000000';
    else
        PSTATE.IT<4:0> = LSL(PSTATE.IT<4:0>, 1);
    return;
```

Library pseudocode for aarch32/functions/system/AArch32.SysRegRead

```
// Read from a 32-bit AArch32 System register and write the register's contents to R[t].
AArch32.SysRegRead(integer cp_num, bits(32) instr, integer t);
```

Library pseudocode for aarch32/functions/system/AArch32.SysRegRead64

```
// Read from a 64-bit AArch32 System register and write the register's contents to R[t] and R[t2].
AArch32.SysRegRead64(integer cp_num, bits(32) instr, integer t, integer t2);
```

Library pseudocode for aarch32/functions/system/AArch32.SysRegReadCanWriteAPSR

```
// AArch32.SysRegReadCanWriteAPSR()
// ================================
// Determines whether the AArch32 System register read instruction can write to APSR flags.

boolean AArch32.SysRegReadCanWriteAPSR(integer cp_num, bits(32) instr)
    assert UsingAArch32();
    assert (cp_num IN {14,15});
    assert cp_num == UInt(instr<11:8>);

    opc1 = UInt(instr<23:21>);
    opc2 = UInt(instr<7:5>);
    CRn  = UInt(instr<19:16>);
    CRm  = UInt(instr<3:0>);

    if cp_num == 14 && opc1 == 0 && CRn == 0 && CRm == 1 && opc2 == 0 then // DBGDSCRint
        return TRUE;

    return FALSE;
```

Library pseudocode for aarch32/functions/system/AArch32.SysRegWrite

```
// Read the contents of R[t] and write to a 32-bit AArch32 System register.
AArch32.SysRegWrite(integer cp_num, bits(32) instr, integer t);
```

Library pseudocode for aarch32/functions/system/AArch32.SysRegWrite64

```
// Read the contents of R[t] and R[t2] and write to a 64-bit AArch32 System register.
AArch32.SysRegWrite64(integer cp_num, bits(32) instr, integer t, integer t2);
```

Library pseudocode for aarch32/functions/system/AArch32.SysRegWriteM

```
// Read a value from a virtual address and write it to an AArch32 System register.
AArch32.SysRegWriteM(integer cp_num, bits(32) instr, bits(32) address);
```

Library pseudocode for aarch32/functions/system/AArch32.WriteMode

```
// AArch32.WriteMode()
// ===================
// Function for dealing with writes to PSTATE.M from AArch32 state only.
// This ensures that PSTATE.EL and PSTATE.SP are always valid.

AArch32.WriteMode(bits(5) mode)
    (valid,el) = ELFromM32(mode);
    assert valid;
    PSTATE.M  = mode;
    PSTATE.EL  = el;
    PSTATE.nRW = '1';
    PSTATE.SP  = (if mode IN {M32_User,M32_System} then '0' else '1');
    return;
```

Library pseudocode for aarch32/functions/system/AArch32.WriteModeByInstr

```
// AArch32.WriteModeByInstr()
// ==========================
// Function for dealing with writes to PSTATE.M from an AArch32 instruction, and ensuring that
// illegal state changes are correctly flagged in PSTATE.IL.

AArch32.WriteModeByInstr(bits(5) mode)
    (valid,el) = ELFromM32(mode);

    // 'valid' is set to FALSE if' mode' is invalid for this implementation or the current value
    // of SCR.NS/SCR_EL3.NS. Additionally, it is illegal for an instruction to write 'mode' to
    // PSTATE.EL if it would result in any of:
    // * A change to a mode that would cause entry to a higher Exception level.
    if UInt(el) > UInt(PSTATE.EL) then
        valid = FALSE;

    // * A change to or from Hyp mode.
    if (PSTATE.M == M32_Hyp || mode == M32_Hyp) && PSTATE.M != mode then
        valid = FALSE;

    // * When EL2 is implemented, the value of HCR.TGE is '1', a change to a Non-secure EL1 mode.
    if PSTATE.M == M32_Monitor && HaveEL(EL2) && el == EL1 && SCR.NS == '1' && HCR.TGE == '1' then
        valid = FALSE;

    if !valid then
        PSTATE.IL = '1';
    else
        AArch32.WriteMode(mode);
```

Library pseudocode for aarch32/functions/system/BadMode

```
// BadMode()
// =========

boolean BadMode(bits(5) mode)
    // Return TRUE if 'mode' encodes a mode that is not valid for this implementation
    boolean valid;
    case mode of
        when M32_Monitor
            valid = HaveAArch32EL(EL3);
        when M32_Hyp
            valid = HaveAArch32EL(EL2);
        when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System
            // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
            // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
            // AArch64, then these modes are EL1 modes.
            // Therefore it is sufficient to test this implementation supports EL1 using AArch32.
            valid = HaveAArch32EL(EL1);
        when M32_User
            valid = HaveAArch32EL(EL0);
        otherwise
            valid = FALSE;          // Passed an illegal mode value
    return !valid;
```

Library pseudocode for aarch32/functions/system/BankedRegisterAccessValid

```
// BankedRegisterAccessValid()
// ===========================
// Checks for MRS (Banked register) or MSR (Banked register) accesses to registers
// other than the SPSRs that are invalid. This includes ELR_hyp accesses.

BankedRegisterAccessValid(bits(5) SYSm, bits(5) mode)

    case SYSm of
        when '000xx', '00100'                           // R8_usr to R12_usr
            if mode != M32_FIQ then UNPREDICTABLE;
        when '00101'                                    // SP_usr
            if mode == M32_System then UNPREDICTABLE;
        when '00110'                                    // LR_usr
            if mode IN {M32_Hyp,M32_System} then UNPREDICTABLE;
        when '010xx', '0110x', '01110'                  // R8_fiq to R12_fiq, SP_fiq, LR_fiq
            if mode == M32_FIQ then UNPREDICTABLE;
        when '1000x'                                    // LR_irq, SP_irq
            if mode == M32_IRQ then UNPREDICTABLE;
        when '1001x'                                    // LR_svc, SP_svc
            if mode == M32_Svc then UNPREDICTABLE;
        when '1010x'                                    // LR_abt, SP_abt
            if mode == M32_Abort then UNPREDICTABLE;
        when '1011x'                                    // LR_und, SP_und
            if mode == M32_Undef then UNPREDICTABLE;
        when '1110x'                                    // LR_mon, SP_mon
            if (!HaveEL(EL3) || CurrentSecurityState() != SS_Secure ||
                mode == M32_Monitor) then UNPREDICTABLE;
        when '11110'                                    // ELR_hyp, only from Monitor or Hyp mode
            if !HaveEL(EL2) || !(mode IN {M32_Monitor,M32_Hyp}) then UNPREDICTABLE;
        when '11111'                                    // SP_hyp, only from Monitor mode
            if !HaveEL(EL2) || mode != M32_Monitor then UNPREDICTABLE;
        otherwise
            UNPREDICTABLE;

    return;
```

Library pseudocode for aarch32/functions/system/CPSRWriteByInstr

```
// CPSRWriteByInstr()
// ==================
// Update PSTATE.<N,Z,C,V,Q,GE,E,A,I,F,M> from a CPSR value written by an MSR instruction.

CPSRWriteByInstr(bits(32) value, bits(4) bytemask)
    privileged = PSTATE.EL != EL0;              // PSTATE.<A,I,F,M> are not writable at EL0

    // Write PSTATE from 'value', ignoring bytes masked by 'bytemask'
    if bytemask<3> == '1' then
        PSTATE.<N,Z,C,V,Q> = value<31:27>;
        // Bits <26:24> are ignored

    if bytemask<2> == '1' then
        if HaveSSBSExt() then
            PSTATE.SSBS = value<23>;
        if privileged then
            PSTATE.PAN = value<22>;
        if HaveDITExt() then
            PSTATE.DIT = value<21>;
        // Bit <20> is RES0
        PSTATE.GE = value<19:16>;

    if bytemask<1> == '1' then
        // Bits <15:10> are RES0
        PSTATE.E = value<9>;                    // PSTATE.E is writable at EL0
        if privileged then
            PSTATE.A = value<8>;

    if bytemask<0> == '1' then
        if privileged then
            PSTATE.<I,F> = value<7:6>;
            // Bit <5> is RES0
            // AArch32.WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
            AArch32.WriteModeByInstr(value<4:0>);
    return;
```

Library pseudocode for aarch32/functions/system/ConditionPassed

```
// ConditionPassed()
// =================

boolean ConditionPassed()
    return ConditionHolds(AArch32.CurrentCond());
```

Library pseudocode for aarch32/functions/system/CurrentCond

```
bits(4) AArch32.CurrentCond();
```

Library pseudocode for aarch32/functions/system/InITBlock

```
// InITBlock()
// ===========

boolean InITBlock()
    if CurrentInstrSet() == InstrSet_T32 then
        return PSTATE.IT<3:0> != '0000';
    else
        return FALSE;
```

Library pseudocode for aarch32/functions/system/LastInITBlock

```
// LastInITBlock()
// ===============

boolean LastInITBlock()
    return (PSTATE.IT<3:0> == '1000');
```

Library pseudocode for aarch32/functions/system/SPSRWriteByInstr

```
// SPSRWriteByInstr()
// ==================

SPSRWriteByInstr(bits(32) value, bits(4) bytemask)

    bits(32) new_spsr = SPSR[];

    if bytemask<3> == '1' then
        new_spsr<31:24> = value<31:24>;  // N,Z,C,V,Q flags, IT[1:0],J bits

    if bytemask<2> == '1' then
        new_spsr<23:16> = value<23:16>;  // IL bit, GE[3:0] flags

    if bytemask<1> == '1' then
        new_spsr<15:8> = value<15:8>;    // IT[7:2] bits, E bit, A interrupt mask

    if bytemask<0> == '1' then
        new_spsr<7:0> = value<7:0>;      // I,F interrupt masks, T bit, Mode bits

    SPSR[] = new_spsr;                   // UNPREDICTABLE if User or System mode

    return;
```

Library pseudocode for aarch32/functions/system/SPSRaccessValid

```
// SPSRaccessValid()
// =================
// Checks for MRS (Banked register) or MSR (Banked register) accesses to the SPSRs
// that are UNPREDICTABLE

SPSRaccessValid(bits(5) SYSm, bits(5) mode)
    case SYSm of
        when '01110'                                                    // SPSR_fiq
            if mode == M32_FIQ   then UNPREDICTABLE;
        when '10000'                                                    // SPSR_irq
            if mode == M32_IRQ   then UNPREDICTABLE;
        when '10010'                                                    // SPSR_svc
            if mode == M32_Svc   then UNPREDICTABLE;
        when '10100'                                                    // SPSR_abt
            if mode == M32_Abort then UNPREDICTABLE;
        when '10110'                                                    // SPSR_und
            if mode == M32_Undef then UNPREDICTABLE;
        when '11100'                                                    // SPSR_mon
            if (!HaveEL(EL3) || mode == M32_Monitor ||
                CurrentSecurityState() != SS_Secure) then UNPREDICTABLE;
        when '11110'                                                    // SPSR_hyp
            if !HaveEL(EL2) || mode != M32_Monitor then UNPREDICTABLE;
        otherwise
            UNPREDICTABLE;

    return;
```

Library pseudocode for aarch32/functions/system/SelectInstrSet

```
// SelectInstrSet()
// ================

SelectInstrSet(InstrSet iset)
    assert CurrentInstrSet() IN {InstrSet_A32, InstrSet_T32};
    assert iset IN {InstrSet_A32, InstrSet_T32};

    PSTATE.T = if iset == InstrSet_A32 then '0' else '1';

    return;
```

Library pseudocode for aarch32/functions/v6simd/Sat

```
// Sat()
// =====

bits(N) Sat(integer i, integer N, boolean unsigned)
    result = if unsigned then UnsignedSat(i, N) else SignedSat(i, N);
    return result;
```

Library pseudocode for aarch32/functions/v6simd/SignedSat

```
// SignedSat()
// ===========

bits(N) SignedSat(integer i, integer N)
    (result, -) = SignedSatQ(i, N);
    return result;
```

Library pseudocode for aarch32/functions/v6simd/UnsignedSat

```
// UnsignedSat()
// =============

bits(N) UnsignedSat(integer i, integer N)
    (result, -) = UnsignedSatQ(i, N);
    return result;
```

Library pseudocode for aarch32/ic/AArch32.IC

```
// AArch32.IC()
// ============
// Perform Instruction Cache Operation.

AArch32.IC(CacheOpScope opscope)
    regval = bits(32) UNKNOWN;
    AArch32.IC(regval, opscope);

// AArch32.IC()
// ============
// Perform Instruction Cache Operation.

AArch32.IC(bits(32) regval, CacheOpScope opscope)
    CacheRecord cache;
    AccType acctype = AccType_IC;

    cache.acctype = acctype;
    cache.cachetype = CacheType_Instruction;
    cache.cacheop = CacheOp_Invalidate;
    cache.opscope = opscope;
    cache.security = SecurityStateAtEL(PSTATE.EL);

    if opscope IN {CacheOpScope_ALLU, CacheOpScope_ALLUIS} then
        if opscope == CacheOpScope_ALLUIS || (opscope == CacheOpScope_ALLU && PSTATE.EL == EL1
            && EL2Enabled() && HCR.FB == '1') then
            cache.shareability = Shareability_ISH;
        else
            cache.shareability = Shareability_NSH;
        cache.regval = ZeroExtend(regval, 64);
        CACHE_OP(cache);
    else
        assert opscope == CacheOpScope_PoU;

        if EL2Enabled() then
            if PSTATE.EL IN {EL0, EL1} then
                cache.is_vmid_valid = TRUE;
                cache.vmid          = VMID[];
            else
                cache.is_vmid_valid = FALSE;
        else
            cache.is_vmid_valid = FALSE;

        if PSTATE.EL == EL0 then
            cache.is_asid_valid = TRUE;
            cache.asid          = ASID[];
        else
            cache.is_asid_valid = FALSE;

        need_translate = ICInstNeedsTranslation(opscope);

        cache.shareability = Shareability_NSH;
        cache.vaddress = ZeroExtend(regval, 64);
        cache.translated = need_translate;

        if !need_translate then
            cache.paddress = FullAddress UNKNOWN;
            CACHE_OP(cache);
            return;

        wasaligned = TRUE;
        iswrite = FALSE;
        size = 0;
        memaddrdesc = AArch32.TranslateAddress(regval, acctype, iswrite, wasaligned, size);
        if IsFault(memaddrdesc) then
            AArch32.Abort(regval, memaddrdesc.fault);

        cache.paddress = memaddrdesc.paddress;
        CACHE_OP(cache);
    return;
```

Library pseudocode for aarch32/predictionrestrict/RestrictPrediction

```
// RestrictPrediction()
// ====================
// Clear all predictions in the context.

AArch32.RestrictPrediction(bits(32) val, RestrictType restriction)

    ExecutionCntxt c;
    target_el    = val<25:24>;

    // If the instruction is executed at an EL lower than the specified
    // level, it is treated as a NOP.
    if UInt(target_el) > UInt(PSTATE.EL) then return;

    bit ns  = val<26>;
    bit nse = bit UNKNOWN;
    ss = TargetSecurityState(ns, nse);

    c.security  = ss;
    c.target_el = target_el;

    if EL2Enabled() then
        if PSTATE.EL IN {EL0, EL1} then
            c.is_vmid_valid = TRUE;
            c.all_vmid      = FALSE;
            c.vmid          = VMID[];

        elsif target_el IN {EL0, EL1} then
            c.is_vmid_valid = TRUE;
            c.all_vmid      = val<27> == '1';
            c.vmid          = ZeroExtend(val<23:16>, 16);      // Only valid if  val<27> == '0';
        else
            c.is_vmid_valid = FALSE;
    else
        c.is_vmid_valid = FALSE;

    if PSTATE.EL == EL0 then
        c.is_asid_valid = TRUE;
        c.all_asid      = FALSE;
        c.asid          = ASID[];

    elsif target_el == EL0 then
        c.is_asid_valid = TRUE;
        c.all_asid      = val<8> == '1';
        c.asid          = ZeroExtend(val<7:0>, 16);            // Only valid if  val<8> == '0';

    else
        c.is_asid_valid = FALSE;

    c.restriction = restriction;
    RESTRICT_PREDICTIONS(c);
```

Library pseudocode for aarch32/translation/attrs/AArch32.DefaultTEXDecode

```
// AArch32.DefaultTEXDecode()
// ==========================
// Apply short-descriptor format memory region attributes, without TEX remap

MemoryAttributes AArch32.DefaultTEXDecode(bits(3) TEX_in, bit C_in, bit B_in, bit s)
    MemoryAttributes mematttrs;
    bits(3) TEX = TEX_in;
    bit C = C_in;
    bit B = B_in;

    // Reserved values map to allocated values
    if (TEX == '001' && C:B == '01') || (TEX == '010' && C:B != '00') || TEX == '011' then
        bits(5) texcb;
        (-, texcb) = ConstrainUnpredictableBits(Unpredictable_RESTEXCB, 5);
        TEX = texcb<4:2>;  C = texcb<1>;  B = texcb<0>;

    // Distinction between Inner Shareable and Outer Shareable is not supported in this format
    // A memory region is either Non-shareable or Outer Shareable
    case TEX:C:B of
        when '00000'
            // Device-nGnRnE
            mematttrs.memtype      = MemType_Device;
            mematttrs.device       = DeviceType_nGnRnE;
            mematttrs.shareability = Shareability_OSH;
        when '00001', '01000'
            // Device-nGnRE
            mematttrs.memtype      = MemType_Device;
            mematttrs.device       = DeviceType_nGnRE;
            mematttrs.shareability = Shareability_OSH;
        when '00010'
            // Write-through Read allocate
            mematttrs.memtype      = MemType_Normal;
            mematttrs.inner.attrs  = MemAttr_WT;
            mematttrs.inner.hints  = MemHint_RA;
            mematttrs.outer.attrs  = MemAttr_WT;
            mematttrs.outer.hints  = MemHint_RA;
            mematttrs.shareability = if s == '1' then Shareability_OSH else Shareability_NSH;
        when '00011'
            // Write-back Read allocate
            mematttrs.memtype      = MemType_Normal;
            mematttrs.inner.attrs  = MemAttr_WB;
            mematttrs.inner.hints  = MemHint_RA;
            mematttrs.outer.attrs  = MemAttr_WB;
            mematttrs.outer.hints  = MemHint_RA;
            mematttrs.shareability = if s == '1' then Shareability_OSH else Shareability_NSH;
        when '00100'
            // Non-cacheable
            mematttrs.memtype      = MemType_Normal;
            mematttrs.inner.attrs  = MemAttr_NC;
            mematttrs.outer.attrs  = MemAttr_NC;
            mematttrs.shareability = Shareability_OSH;
        when '00110'
            mematttrs = MemoryAttributes IMPLEMENTATION_DEFINED;
        when '00111'
            // Write-back Read and Write allocate
            mematttrs.memtype      = MemType_Normal;
            mematttrs.inner.attrs  = MemAttr_WB;
            mematttrs.inner.hints  = MemHint_RWA;
            mematttrs.outer.attrs  = MemAttr_WB;
            mematttrs.outer.hints  = MemHint_RWA;
            mematttrs.shareability = if s == '1' then Shareability_OSH else Shareability_NSH;
        when '1xxxx'
            // Cacheable, TEX<1:0> = Outer attrs, {C,B} = Inner attrs
            mematttrs.memtype = MemType_Normal;
            mematttrs.inner   = DecodeSDFAttr(C:B);
            mematttrs.outer   = DecodeSDFAttr(TEX<1:0>);

            if mematttrs.inner.attrs == MemAttr_NC && mematttrs.outer.attrs == MemAttr_NC then
                mematttrs.shareability = Shareability_OSH;
            else
```

```
            mematttrs.shareability = if s == '1' then Shareability_OSH else Shareability_NSH;
        otherwise
            // Reserved, handled above
            Unreachable();

    // The Transient hint is not supported in this format
    mematttrs.inner.transient = FALSE;
    mematttrs.outer.transient = FALSE;
    mematttrs.tagged          = FALSE;

    if mematttrs.inner.attrs == MemAttr_WB && mematttrs.outer.attrs == MemAttr_WB then
        mematttrs.xs = '0';
    else
        mematttrs.xs = '1';

    return mematttrs;
```

Library pseudocode for aarch32/translation/attrs/AArch32.RemappedTEXDecode

```
// AArch32.RemappedTEXDecode()
// ===========================
// Apply short-descriptor format memory region attributes, with TEX remap

MemoryAttributes AArch32.RemappedTEXDecode(Regime regime, bits(3) TEX, bit C, bit B, bit s)

    MemoryAttributes memattrs;
    PRRR_Type prrr;
    NMRR_Type nmrr;

    region = UInt(TEX<0>:C:B);         // TEX<2:1> are ignored in this mapping scheme
    if region == 6 then
        return MemoryAttributes IMPLEMENTATION_DEFINED;

    if regime == Regime_EL30 then
        prrr = PRRR_S;
        nmrr = NMRR_S;
    elsif HaveAArch32EL(EL3) then
        prrr = PRRR_NS;
        nmrr = NMRR_NS;
    else
        prrr = PRRR;
        nmrr = NMRR;

    base = 2 * region;
    attrfield = prrr<base+1:base>;

    if attrfield == '11' then      // Reserved, maps to allocated value
        (-, attrfield) = ConstrainUnpredictableBits(Unpredictable_RESPRRR, 2);

    case attrfield of
        when '00'                   // Device-nGnRnE
            memattrs.memtype      = MemType_Device;
            memattrs.device       = DeviceType_nGnRnE;
            memattrs.shareability = Shareability_OSH;
        when '01'                   // Device-nGnRE
            memattrs.memtype      = MemType_Device;
            memattrs.device       = DeviceType_nGnRE;
            memattrs.shareability = Shareability_OSH;
        when '10'
            NSn  = if s == '0' then prrr.NS0 else prrr.NS1;
            NOSm = prrr<region+24> AND NSn;
            IRn  = nmrr<base+1:base>;
            ORn  = nmrr<base+17:base+16>;

            memattrs.memtype = MemType_Normal;
            memattrs.inner   = DecodeSDFAttr(IRn);
            memattrs.outer   = DecodeSDFAttr(ORn);
            if memattrs.inner.attrs == MemAttr_NC && memattrs.outer.attrs == MemAttr_NC then
                memattrs.shareability = Shareability_OSH;
            else
                bits(2) sh = NSn:NOSm;
                memattrs.shareability = DecodeShareability(sh);
        when '11'
            Unreachable();

    // The Transient hint is not supported in this format
    memattrs.inner.transient = FALSE;
    memattrs.outer.transient = FALSE;
    memattrs.tagged          = FALSE;

    if memattrs.inner.attrs == MemAttr_WB && memattrs.outer.attrs == MemAttr_WB then
        memattrs.xs = '0';
    else
        memattrs.xs = '1';

    return memattrs;
```

Library pseudocode for aarch32/translation/debug/AArch32.CheckBreakpoint

```
// AArch32.CheckBreakpoint()
// =========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime, when either debug exceptions are enabled, or halting debug is enabled
// and halting is allowed.

FaultRecord AArch32.CheckBreakpoint(bits(32) vaddress,  integer size)
    assert ELUsingAArch32(S1TranslationRegime());
    assert size IN {2,4};

    match = FALSE;
    mismatch = FALSE;

    for i = 0 to NumBreakpointsImplemented() - 1
        (match_i, mismatch_i) = AArch32.BreakpointMatch(i, vaddress, size);
        match = match || match_i;
        mismatch = mismatch || mismatch_i;

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Breakpoint;
        Halt(reason);
    elsif (match || mismatch) then
        acctype = AccType_IFETCH;
        iswrite = FALSE;
        debugmoe = DebugException_Breakpoint;
        return AArch32.DebugFault(acctype, iswrite, debugmoe);
    else
        return NoFault();
```

Library pseudocode for aarch32/translation/debug/AArch32.CheckDebug

```
// AArch32.CheckDebug()
// ====================
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch32.CheckDebug(bits(32) vaddress, AccType acctype, boolean iswrite, integer size)

    FaultRecord fault = NoFault();

    boolean d_side = (IsDataAccess(acctype) || acctype == AccType_DC);
    boolean i_side = (acctype == AccType_IFETCH);
    generate_exception = AArch32.GenerateDebugExceptions() && DBGDSCRext.MDBGen == '1';
    halt = HaltOnBreakpointOrWatchpoint();
    // Relative priority of Vector Catch and Breakpoint exceptions not defined in the architecture
    vector_catch_first = ConstrainUnpredictableBool(Unpredictable_BPVECTORCATCHPRI);

    if i_side && vector_catch_first && generate_exception then
        fault = AArch32.CheckVectorCatch(vaddress, size);

    if fault.statuscode == Fault_None && (generate_exception || halt) then
        if d_side then
            fault = AArch32.CheckWatchpoint(vaddress, acctype, iswrite, size);
        elsif i_side then
            fault = AArch32.CheckBreakpoint(vaddress, size);

    if fault.statuscode == Fault_None && i_side && !vector_catch_first && generate_exception then
        return AArch32.CheckVectorCatch(vaddress, size);

    return fault;
```

Library pseudocode for aarch32/translation/debug/AArch32.CheckVectorCatch

```
// AArch32.CheckVectorCatch()
// ==========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch32
// translation regime, when debug exceptions are enabled.

FaultRecord AArch32.CheckVectorCatch(bits(32) vaddress, integer size)
    assert ELUsingAArch32(S1TranslationRegime());

    match = AArch32.VCRMatch(vaddress);
    if size == 4 && !match && AArch32.VCRMatch(vaddress + 2) then
        match = ConstrainUnpredictableBool(Unpredictable_VCMATCHHALF);

    if match then
        acctype = AccType_IFETCH;
        iswrite = FALSE;
        debugmoe = DebugException_VectorCatch;
        return AArch32.DebugFault(acctype, iswrite, debugmoe);
    else
        return NoFault();
```

Library pseudocode for aarch32/translation/debug/AArch32.CheckWatchpoint

```
// AArch32.CheckWatchpoint()
// =========================
// Called before accessing the memory location of "size" bytes at "address",
// when either debug exceptions are enabled for the access, or halting debug
// is enabled and halting is allowed.

FaultRecord AArch32.CheckWatchpoint(bits(32) vaddress, AccType acctype,
                                    boolean iswrite, integer size)
    assert ELUsingAArch32(S1TranslationRegime());

    if acctype == AccType_DC then
        if !iswrite then
            return NoFault();
        elsif !(boolean IMPLEMENTATION_DEFINED "DCIMVAC generates watchpoint") then
            return NoFault();
    elsif !IsDataAccess(acctype) then
        return NoFault();

    match = FALSE;
    ispriv = AArch32.AccessUsesEL(acctype) != EL0;

    for i = 0 to NumWatchpointsImplemented() - 1
        if AArch32.WatchpointMatch(i, vaddress, size, ispriv, acctype, iswrite) then
            match = TRUE;

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Watchpoint;
        EDWAR = ZeroExtend(vaddress, 64);
        Halt(reason);
    elsif match then
        debugmoe = DebugException_Watchpoint;
        return AArch32.DebugFault(acctype, iswrite, debugmoe);
    else
        return NoFault();
```

Library pseudocode for aarch32/translation/faults/AArch32.DebugFault

```
// AArch32.DebugFault()
// ====================
// Return a fault record indicating a hardware watchpoint/breakpoint

FaultRecord AArch32.DebugFault(AccType acctype, boolean iswrite, bits(4) debugmoe)
    FaultRecord fault;

    fault.statuscode  = Fault_Debug;
    fault.acctype     = acctype;
    fault.write       = iswrite;
    fault.debugmoe    = debugmoe;
    fault.secondstage = FALSE;
    fault.s2fs1walk   = FALSE;

    return fault;
```

Library pseudocode for aarch32/translation/faults/AArch32.IPAIsOutOfRange

```
// AArch32.IPAIsOutOfRange()
// =========================
// Check intermediate physical address bits not resolved by translation are ZERO

boolean AArch32.IPAIsOutOfRange(S2TTWParams walkparams, bits(40) ipa)
    // Input Address size
    iasize = AArch32.S2IASize(walkparams.t0sz);

    return iasize < 40 && !IsZero(ipa<39:iasize>);
```

Library pseudocode for aarch32/translation/faults/AArch32.S1HasAlignmentFault

```
// AArch32.S1HasAlignmentFault()
// =============================
// Returns whether stage 1 output fails alignment requirement on data accesses
// to Device memory

boolean AArch32.S1HasAlignmentFault(AccType acctype, boolean aligned,
                                    bit ntlsmd, MemoryAttributes memattrs)
    if acctype == AccType_IFETCH || memattrs.memtype != MemType_Device then
        return FALSE;

    if acctype == AccType_A32LSMD && ntlsmd == '0' && memattrs.device != DeviceType_GRE then
        return TRUE;

    return !aligned || acctype == AccType_DCZVA;
```

Library pseudocode for aarch32/translation/faults/AArch32.S1LDHasPermissionsFault

```
// AArch32.S1LDHasPermissionsFault()
// =================================
// Returns whether an access using stage 1 long-descriptor translation
// violates permissions of target memory

boolean AArch32.S1LDHasPermissionsFault(Regime regime, SecurityState ss, S1TTWParams walkparams,
                                        Permissions perms, MemType memtype, PASpace paspace,
                                        boolean ispriv, AccType acctype, boolean iswrite)
    bit r;
    bit w;
    bit x;
    bit pr;
    bit pw;
    bit ur;
    bit uw;
    bit xn;
    if HasUnprivileged(regime) then
        // Apply leaf permissions
        case perms.ap<2:1> of
            when '00' (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1 only
            when '01' (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any PL
            when '10' (pr,pw,ur,uw) = ('1','0','0','0'); // RO at PL1 only
            when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL

        // Apply hierarchical permissions
        case perms.ap_table of
            when '00' (pr,pw,ur,uw) = ( pr, pw, ur, uw); // No effect
            when '01' (pr,pw,ur,uw) = ( pr, pw,'0','0'); // Privileged access
            when '10' (pr,pw,ur,uw) = ( pr,'0', ur,'0'); // Read-only
            when '11' (pr,pw,ur,uw) = ( pr,'0','0','0'); // Read-only, privileged access

        xn   = perms.xn OR perms.xn_table;
        pxn  = perms.pxn OR perms.pxn_table;

        ux = ur AND NOT(xn OR (uw AND walkparams.wxn));
        px = pr AND NOT(xn OR pxn OR (pw AND walkparams.wxn) OR (uw AND walkparams.uwxn));

        pan_access = !(acctype IN {AccType_DC, AccType_IFETCH, AccType_AT});
        if HavePANExt() && pan_access then
            pan = PSTATE.PAN AND (ur OR uw);
            pr  = pr AND NOT(pan);
            pw  = pw AND NOT(pan);

        (r,w,x) = if ispriv then (pr,pw,px) else (ur,uw,ux);

        // Prevent execution from Non-secure space by PE in Secure state if SIF is set
        if ss == SS_Secure && paspace == PAS_NonSecure then
            x = x AND NOT(walkparams.sif);
    else
        // Apply leaf permissions
        case perms.ap<2> of
            when '0' (r,w) = ('1','1'); // No effect
            when '1' (r,w) = ('1','0'); // Read-only

        // Apply hierarchical permissions
        case perms.ap_table<1> of
            when '0' (r,w) = ( r , w ); // No effect
            when '1' (r,w) = ( r ,'0'); // Read-only

        xn = perms.xn OR perms.xn_table;
        x  = NOT(xn OR (w AND walkparams.wxn));

    if acctype == AccType_IFETCH then
        constraint = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
        if constraint == Constraint_FAULT && memtype == MemType_Device then
            return TRUE;
        else
            return x == '0';
    elsif acctype IN {AccType_IC, AccType_DC} then
        return FALSE;
```

```
    elsif iswrite then
        return w == '0';
    else
        return r == '0';
```

Library pseudocode for aarch32/translation/faults/AArch32.S1SDHasPermissionsFault

```
// AArch32.S1SDHasPermissionsFault()
// =================================
// Returns whether an access using stage 1 short-descriptor translation
// violates permissions of target memory

boolean AArch32.S1SDHasPermissionsFault(Regime regime, SecurityState ss, Permissions perms_in,
                                        MemType memtype, PASpace paspace, boolean ispriv,
                                        AccType acctype, boolean iswrite)
    Permissions perms = perms_in;
    bit pr;
    bit pw;
    bit ur;
    bit uw;
    SCTLR_Type sctlr;
    if regime == Regime_EL30 then
        sctlr = SCTLR_S;
    elsif HaveAArch32EL(EL3) then
        sctlr = SCTLR_NS;
    else
        sctlr = SCTLR;

    if sctlr.AFE == '0' then
        // Map Reserved encoding '100'
        if perms.ap == '100' then
            perms.ap = bits(3) IMPLEMENTATION_DEFINED "Reserved short descriptor AP encoding";

        case perms.ap of
            when '000' (pr,pw,ur,uw) = ('0','0','0','0'); // No access
            when '001' (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1 only
            when '010' (pr,pw,ur,uw) = ('1','1','1','0'); // R/W at PL1, RO at PL0
            when '011' (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any PL
            //   '100' is reserved
            when '101' (pr,pw,ur,uw) = ('1','0','0','0'); // RO at PL1 only
            when '110' (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL (deprecated)
            when '111' (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL
    else // Simplified access permissions model
        case perms.ap<2:1> of
            when '00' (pr,pw,ur,uw) = ('1','1','0','0'); // R/W at PL1 only
            when '01' (pr,pw,ur,uw) = ('1','1','1','1'); // R/W at any PL
            when '10' (pr,pw,ur,uw) = ('1','0','0','0'); // RO at PL1 only
            when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // RO at any PL

    ux = ur AND NOT(perms.xn OR (uw AND sctlr.WXN));
    px = pr AND NOT(perms.xn OR perms.pxn OR (pw AND sctlr.WXN) OR (uw AND sctlr.UWXN));

    pan_access = !(acctype IN {AccType_DC, AccType_IFETCH, AccType_AT});
    if HavePANExt() && pan_access then
        pan = PSTATE.PAN AND (ur OR uw);
        pr  = pr AND NOT(pan);
        pw  = pw AND NOT(pan);

    (r,w,x) = if ispriv then (pr,pw,px) else (ur,uw,ux);

    // Prevent execution from Non-secure space by PE in Secure state if SIF is set
    if ss == SS_Secure && paspace == PAS_NonSecure then
        x = x AND NOT(if ELUsingAArch32(EL3) then SCR.SIF else SCR_EL3.SIF);

    if acctype == AccType_IFETCH then
        constraint = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
        if constraint == Constraint_FAULT && memtype == MemType_Device then
            return TRUE;
        else
            return x == '0';
    elsif acctype IN {AccType_IC, AccType_DC} then
        return FALSE;
    elsif iswrite then
        return w == '0';
    else
        return r == '0';
```

Library pseudocode for aarch32/translation/faults/AArch32.S2HasAlignmentFault

```
// AArch32.S2HasAlignmentFault()
// =============================
// Returns whether stage 2 output fails alignment requirement on data accesses
// to Device memory

boolean AArch32.S2HasAlignmentFault(AccType acctype, boolean aligned,
                                    MemoryAttributes memattrs)
    if acctype == AccType_IFETCH || memattrs.memtype != MemType_Device then
        return FALSE;

    return !aligned || acctype == AccType_DCZVA;
```

Library pseudocode for aarch32/translation/faults/AArch32.S2HasPermissionsFault

```
// AArch32.S2HasPermissionsFault()
// ===============================
// Returns whether stage 2 access violates permissions of target memory

boolean AArch32.S2HasPermissionsFault(boolean s2fs1walk, S2TTWParams walkparams,
                                      Permissions perms, MemType memtype,
                                      boolean ispriv, AccType acctype,
                                      boolean iswrite)
    bit px;
    bit ux;
    r = perms.s2ap<0>;
    w = perms.s2ap<1>;
    bit x;
    if HaveExtendedExecuteNeverExt() then
        case perms.s2xn:perms.s2xnx of
            when '00'  (px, ux) = ( r , r );
            when '01'  (px, ux) = ('0', r );
            when '10'  (px, ux) = ('0','0');
            when '11'  (px, ux) = ( r ,'0');

        x = if ispriv then px else ux;
    else
        x = r AND NOT(perms.s2xn);

    if s2fs1walk && walkparams.ptw == '1' && memtype == MemType_Device then
        return TRUE;
    elsif acctype == AccType_IFETCH then
        constraint = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
        if constraint == Constraint_FAULT && memtype == MemType_Device then
            return TRUE;
        else
            return x == '0';
    elsif acctype IN {AccType_IC, AccType_DC} then
        return FALSE;
    elsif iswrite then
        return w == '0';
    else
        return r == '0';
```

Library pseudocode for aarch32/translation/faults/AArch32.S2InconsistentSL

```
// AArch32.S2InconsistentSL()
// ==========================
// Detect inconsistent configuration of stage 2 T0SZ and SL fields

boolean AArch32.S2InconsistentSL(S2TTWParams walkparams)
    startlevel  = AArch32.S2StartLevel(walkparams.sl0);
    levels      = FINAL_LEVEL - startlevel;
    granulebits = TGxGranuleBits(walkparams.tgx);
    stride      = granulebits - 3;

    // Input address size must at least be large enough to be resolved from the start level
    sl_min_iasize = (
        levels * stride // Bits resolved by table walk, except initial level
        + granulebits   // Bits directly mapped to output address
        + 1);           // At least 1 more bit to be decoded by initial level

    // Can accomodate 1 more stride in the level + concatenation of up to 2^4 tables
    sl_max_iasize = sl_min_iasize + (stride-1) + 4;
    // Configured Input Address size
    iasize        = AArch32.S2IASize(walkparams.t0sz);

    return iasize < sl_min_iasize || iasize > sl_max_iasize;
```

Library pseudocode for aarch32/translation/faults/AArch32.VAIsOutOfRange

```
// AArch32.VAIsOutOfRange()
// ========================
// Check virtual address bits not resolved by translation are identical
// and of accepted value

boolean AArch32.VAIsOutOfRange(Regime regime, S1TTWParams walkparams, bits(32) va)
    if regime == Regime_EL2 then
        // Input Address size
        iasize = AArch32.S1IASize(walkparams.t0sz);
        return walkparams.t0sz != '000' && !IsZero(va<31:iasize>);
    elsif walkparams.t1sz != '000' && walkparams.t0sz != '000' then
        // Lower range Input Address size
        lo_iasize = AArch32.S1IASize(walkparams.t0sz);
        // Upper range Input Address size
        up_iasize = AArch32.S1IASize(walkparams.t1sz);
        return !IsZero(va<31:lo_iasize>) && !IsOnes(va<31:up_iasize>);
    else
        return FALSE;
```

Library pseudocode for aarch32/translation/tlbcontext/AArch32.GetS1TLBContext

```
// AArch32.GetS1TLBContext()
// =========================
// Gather translation context for accesses with VA to match against TLB entries

TLBContext AArch32.GetS1TLBContext(Regime regime, SecurityState ss, bits(32) va)
    TLBContext tlbcontext;

    case regime of
        when Regime_EL2  tlbcontext = AArch32.TLBContextEL2(va);
        when Regime_EL10 tlbcontext = AArch32.TLBContextEL10(ss, va);
        when Regime_EL30 tlbcontext = AArch32.TLBContextEL30(va);

    tlbcontext.includes_s1 = TRUE;
    // The following may be amended for EL1&0 Regime if caching of stage 2 is successful
    tlbcontext.includes_s2 = FALSE;
    return tlbcontext;
```

Library pseudocode for aarch32/translation/tlbcontext/AArch32.GetS2TLBContext

```
// AArch32.GetS2TLBContext()
// =========================
// Gather translation context for accesses with IPA to match against TLB entries

TLBContext AArch32.GetS2TLBContext(FullAddress ipa)
    assert ipa.paspace == PAS_NonSecure;

    TLBContext tlbcontext;

    tlbcontext.ss          = SS_NonSecure;
    tlbcontext.regime      = Regime_EL10;
    tlbcontext.ipaspace    = ipa.paspace;
    tlbcontext.vmid        = ZeroExtend(VTTBR.VMID, 16);
    tlbcontext.tg          = TGx_4KB;
    tlbcontext.includes_s1 = FALSE;
    tlbcontext.includes_s2 = TRUE;
    tlbcontext.ia          = ZeroExtend(ipa.address, 64);
    tlbcontext.cnp         = if HaveCommonNotPrivateTransExt() then VTTBR.CnP else '0';

    return tlbcontext;
```

Library pseudocode for aarch32/translation/tlbcontext/AArch32.TLBContextEL10

```
// AArch32.TLBContextEL10()
// ========================
// Gather translation context for accesses under EL10 regime
// (PL10 when EL3 is A64) to match against TLB entries

TLBContext AArch32.TLBContextEL10(SecurityState ss, bits(32) va)
    TLBContext tlbcontext;
    TTBCR_Type ttbcr;
    TTBR0_Type ttbr0;
    TTBR1_Type ttbr1;
    CONTEXTIDR_Type contextidr;

    if HaveAArch32EL(EL3) then
        ttbcr      = TTBCR_NS;
        ttbr0      = TTBR0_NS;
        ttbr1      = TTBR1_NS;
        contextidr = CONTEXTIDR_NS;
    else
        ttbcr      = TTBCR;
        ttbr0      = TTBR0;
        ttbr1      = TTBR1;
        contextidr = CONTEXTIDR;

    tlbcontext.ss     = ss;
    tlbcontext.regime = Regime_EL10;

    if AArch32.EL2Enabled(ss) then
        tlbcontext.vmid = ZeroExtend(VTTBR.VMID, 16);

    if ttbcr.EAE == '1' then
        tlbcontext.asid = ZeroExtend(if ttbcr.A1 == '0' then ttbr0.ASID else ttbr1.ASID, 16);
    else
        tlbcontext.asid = ZeroExtend(contextidr.ASID, 16);

    tlbcontext.tg = TGx_4KB;
    tlbcontext.ia = ZeroExtend(va, 64);

    if HaveCommonNotPrivateTransExt() && ttbcr.EAE == '1' then
        if AArch32.GetVARange(va, ttbcr.T0SZ, ttbcr.T1SZ) == VARange_LOWER then
            tlbcontext.cnp = ttbr0.CnP;
        else
            tlbcontext.cnp = ttbr1.CnP;
    else
        tlbcontext.cnp = '0';

    return tlbcontext;
```

Library pseudocode for aarch32/translation/tlbcontext/AArch32.TLBContextEL2

```
// AArch32.TLBContextEL2()
// =======================
// Gather translation context for accesses under EL2 regime to match against TLB entries

TLBContext AArch32.TLBContextEL2(bits(32) va)
    TLBContext tlbcontext;

    tlbcontext.ss     = SS_NonSecure;
    tlbcontext.regime = Regime_EL2;
    tlbcontext.ia     = ZeroExtend(va, 64);
    tlbcontext.tg     = TGx_4KB;
    tlbcontext.cnp    = if HaveCommonNotPrivateTransExt() then HTTBR.CnP else '0';

    return tlbcontext;
```

Library pseudocode for aarch32/translation/tlbcontext/AArch32.TLBContextEL30

```
// AArch32.TLBContextEL30()
// =======================
// Gather translation context for accesses under EL30 regime
// (PL10 in Secure state and EL3 is A32) to match against TLB entries

TLBContext AArch32.TLBContextEL30(bits(32) va)
    TLBContext tlbcontext;

    tlbcontext.ss     = SS_Secure;
    tlbcontext.regime = Regime_EL30;

    if TTBCR_S.EAE == '1' then
        tlbcontext.asid = ZeroExtend(if TTBCR_S.A1 == '0' then TTBR0_S.ASID else TTBR1_S.ASID, 16);
    else
        tlbcontext.asid = ZeroExtend(CONTEXTIDR_S.ASID, 16);

    tlbcontext.tg = TGx_4KB;
    tlbcontext.ia = ZeroExtend(va, 64);

    if HaveCommonNotPrivateTransExt() && TTBCR_S.EAE == '1' then
        if AArch32.GetVARange(va, TTBCR_S.T0SZ, TTBCR_S.T1SZ) == VARange_LOWER then
            tlbcontext.cnp = TTBR0_S.CnP;
        else
            tlbcontext.cnp = TTBR1_S.CnP;
    else
        tlbcontext.cnp = '0';

    return tlbcontext;
```

Library pseudocode for aarch32/translation/translation/AArch32.AccessUsesEL

```
// AArch32.AccessUsesEL()
// ======================
// Determine the privilege associated with the access

bits(2) AArch32.AccessUsesEL(AccType acctype)
    if acctype == AccType_UNPRIV then
        return EL0;
    else
        return PSTATE.EL;
```

Library pseudocode for aarch32/translation/translation/AArch32.EL2Enabled

```
// AArch32.EL2Enabled()
// ====================
// Returns whether EL2 is enabled for the given Security State

boolean AArch32.EL2Enabled(SecurityState ss)
    if ss == SS_Secure then
        if !(HaveEL(EL2) && HaveSecureEL2Ext()) then
            return FALSE;
        elsif HaveEL(EL3) then
            return SCR_EL3.EEL2 == '1';
        else
            return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";
    else
        return HaveEL(EL2);
```

Library pseudocode for aarch32/translation/translation/AArch32.FullTranslate

```
// AArch32.FullTranslate()
// =======================
// Perform address translation as specified by VMSA-A32

AddressDescriptor AArch32.FullTranslate(bits(32) va, AccType acctype,
                                        boolean iswrite, boolean aligned)

    // Prepare fault fields in case a fault is detected
    fault = NoFault();
    fault.acctype = acctype;
    fault.write   = iswrite;

    regime = TranslationRegime(PSTATE.EL, acctype);
    ispriv = PSTATE.EL != EL0 && acctype != AccType_UNPRIV;
    ss = SecurityStateForRegime(regime);

    // First Stage Translation
    AddressDescriptor ipa;
    if regime == Regime_EL2 || TTBCR.EAE == '1' then
        (fault, ipa) = AArch32.S1TranslateLD(fault, regime, ss, va, acctype,
                                             aligned, iswrite, ispriv);
    else
        (fault, ipa, -) = AArch32.S1TranslateSD(fault, regime, ss, va, acctype,
                                                aligned, iswrite, ispriv);

    if fault.statuscode != Fault_None then
        return CreateFaultyAddressDescriptor(ZeroExtend(va, 64), fault);

    if regime == Regime_EL10 && EL2Enabled() then
        ipa.vaddress = ZeroExtend(va, 64);
        s2fs1walk = FALSE;
        AddressDescriptor pa;
        (fault, pa) = AArch32.S2Translate(fault, ipa, ss, s2fs1walk, acctype,
                                          aligned, iswrite, ispriv);

        if fault.statuscode != Fault_None then
            return CreateFaultyAddressDescriptor(ZeroExtend(va, 64), fault);
        else
            return pa;
    else
        return ipa;
```

Library pseudocode for aarch32/translation/translation/AArch32.OutputDomain

```
// AArch32.OutputDomain()
// ======================
// Determine the domain the translated output address

bits(2) AArch32.OutputDomain(Regime regime, bits(4) domain)
    bits(2) Dn;
    index = 2 * UInt(domain);
    if regime == Regime_EL30 then
        Dn = DACR_S<index+1:index>;
    elsif HaveAArch32EL(EL3) then
        Dn = DACR_NS<index+1:index>;
    else
        Dn = DACR<index+1:index>;

    if Dn == '10' then
        // Reserved value maps to an allocated value
        (-, Dn) = ConstrainUnpredictableBits(Unpredictable_RESDACR, 2);

    return Dn;
```

Library pseudocode for aarch32/translation/translation/AArch32.S1DisabledOutput

```
// AArch32.S1DisabledOutput()
// ==========================
// Flat map the VA to IPA/PA, depending on the regime, assigning default memory attributes

(FaultRecord, AddressDescriptor) AArch32.S1DisabledOutput(FaultRecord fault_in, Regime regime,
                                                          SecurityState ss, bits(32) va,
                                                          AccType acctype, boolean aligned)

    FaultRecord fault = fault_in;
    // No memory page is guarded when stage 1 address translation is disabled
    SetInGuardedPage(FALSE);

    MemoryAttributes memattrs;
    bit default_cacheable;
    if regime == Regime_EL10 && AArch32.EL2Enabled(ss) then
        if ELStateUsingAArch32(EL2, ss == SS_Secure) then
            default_cacheable = HCR.DC;
        else
            default_cacheable = HCR_EL2.DC;
    else
        default_cacheable = '0';

    if default_cacheable == '1' then
        // Use default cacheable settings
        memattrs.memtype      = MemType_Normal;
        memattrs.inner.attrs  = MemAttr_WB;
        memattrs.inner.hints  = MemHint_RWA;
        memattrs.outer.attrs  = MemAttr_WB;
        memattrs.outer.hints  = MemHint_RWA;
        memattrs.shareability = Shareability_NSH;
        if !ELStateUsingAArch32(EL2, ss == SS_Secure) && HaveMTE2Ext() then
            memattrs.tagged = HCR_EL2.DCT == '1';
        else
            memattrs.tagged = FALSE;
    elsif acctype == AccType_IFETCH then
        memattrs.memtype      = MemType_Normal;
        memattrs.shareability = Shareability_OSH;
        memattrs.tagged       = FALSE;
        if AArch32.S1ICacheEnabled(regime) then
            memattrs.inner.attrs = MemAttr_WT;
            memattrs.inner.hints = MemHint_RA;
            memattrs.outer.attrs = MemAttr_WT;
            memattrs.outer.hints = MemHint_RA;
        else
            memattrs.inner.attrs = MemAttr_NC;
            memattrs.outer.attrs = MemAttr_NC;
    else
        // Treat memory region as Device
        memattrs.memtype      = MemType_Device;
        memattrs.device       = DeviceType_nGnRnE;
        memattrs.shareability = Shareability_OSH;
        memattrs.tagged       = FALSE;

    bit ntlsmd;
    if HaveTrapLoadStoreMultipleDeviceExt() then
        case regime of
            when Regime_EL30 ntlsmd = SCTLR_S.nTLSMD;
            when Regime_EL2  ntlsmd = HSCTLR.nTLSMD;
            when Regime_EL10 ntlsmd = if HaveAArch32EL(EL3) then SCTLR_NS.nTLSMD else SCTLR.nTLSMD;
    else
        ntlsmd = '1';

    if AArch32.S1HasAlignmentFault(acctype, aligned, ntlsmd, memattrs) then
        fault.statuscode  = Fault_Alignment;
        return (fault, AddressDescriptor UNKNOWN);

    FullAddress oa;
    oa.address = ZeroExtend(va, 52);
    oa.paspace = if ss == SS_Secure then PAS_Secure else PAS_NonSecure;
    ipa = CreateAddressDescriptor(ZeroExtend(va, 64), oa, memattrs);
```

```
    return (fault, ipa);
```

Library pseudocode for aarch32/translation/translation/AArch32.S1Enabled

```
// AArch32.S1Enabled()
// ===================
// Returns whether stage 1 translation is enabled for the active translation regime

boolean AArch32.S1Enabled(Regime regime, SecurityState ss)
    if regime == Regime_EL2 then
        return HSCTLR.M == '1';
    elsif regime == Regime_EL30 then
        return SCTLR_S.M == '1';
    elsif !AArch32.EL2Enabled(ss) then
        return (if HaveAArch32EL(EL3) then SCTLR_NS.M else SCTLR.M) == '1';
    elsif ELStateUsingAArch32(EL2, ss == SS_Secure) then
        return HCR.<TGE,DC> == '00' && (if HaveAArch32EL(EL3) then SCTLR_NS.M else SCTLR.M) == '1';
    else
        return HCR_EL2.<TGE,DC> == '00' && SCTLR.M == '1';
```

Library pseudocode for aarch32/translation/translation/AArch32.S1TranslateLD

```
// AArch32.S1TranslateLD()
// =======================
// Perform a stage 1 translation using long-descriptor format mapping VA to IPA/PA
// depending on the regime

(FaultRecord, AddressDescriptor) AArch32.S1TranslateLD(FaultRecord fault_in, Regime regime,
                                                       SecurityState ss, bits(32) va,
                                                       AccType acctype, boolean aligned,
                                                       boolean iswrite, boolean ispriv)
    FaultRecord fault = fault_in;
    fault.secondstage = FALSE;
    fault.s2fs1walk   = FALSE;

    if !AArch32.S1Enabled(regime, ss) then
        return AArch32.S1DisabledOutput(fault, regime, ss, va, acctype, aligned);

    walkparams = AArch32.GetS1TTWParams(regime, va);

    if AArch32.VAIsOutOfRange(regime, walkparams, va) then
        fault.level      = 1;
        fault.statuscode = Fault_Translation;
        return (fault, AddressDescriptor UNKNOWN);

    TTWState walkstate;
    (fault, walkstate) = AArch32.S1WalkLD(fault, regime, ss, walkparams, va, ispriv);

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);

    SetInGuardedPage(FALSE);  // AArch32-VMSA does not guard any pages

    if AArch32.S1HasAlignmentFault(acctype, aligned, walkparams.ntlsmd, walkstate.mematTrs) then
        fault.statuscode = Fault_Alignment;
    elsif IsAtomicRW(acctype) then
        if AArch32.S1LDHasPermissionsFault(regime, ss, walkparams,
                                           walkstate.permissions,
                                           walkstate.mematTrs.memtype,
                                           walkstate.baseaddress.paspace,
                                           ispriv, acctype, FALSE) then
            // The Permission fault was not caused by lack of write permissions
            fault.statuscode = Fault_Permission;
            fault.write      = FALSE;
        elsif AArch32.S1LDHasPermissionsFault(regime, ss, walkparams,
                                              walkstate.permissions,
                                              walkstate.mematTrs.memtype,
                                              walkstate.baseaddress.paspace,
                                              ispriv, acctype, TRUE) then
            // The Permission fault was caused by lack of write permissions
            fault.statuscode = Fault_Permission;
            fault.write      = TRUE;
    elsif AArch32.S1LDHasPermissionsFault(regime, ss, walkparams,
                                          walkstate.permissions,
                                          walkstate.mematTrs.memtype,
                                          walkstate.baseaddress.paspace,
                                          ispriv, acctype, iswrite) then
        fault.statuscode = Fault_Permission;

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);

    MemoryAttributes mematTrs;
    if ((acctype == AccType_IFETCH &&
            (walkstate.mematTrs.memtype == MemType_Device || !AArch32.S1ICacheEnabled(regime))) ||
        (acctype != AccType_IFETCH &&
             walkstate.mematTrs.memtype == MemType_Normal && !AArch32.S1DCacheEnabled(regime))) then
        // Treat memory attributes as Normal Non-Cacheable
        mematTrs = NormalNCMemAttr();
        mematTrs.xs = walkstate.mematTrs.xs;
    else
        mematTrs = walkstate.mematTrs;
```

```
// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime_EL10 && AArch32.EL2Enabled(ss) &&
        (if ELStateUsingAArch32(EL2, ss == SS_Secure) then HCR.VM else HCR_EL2.VM) == '1' &&
        !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
    mattrs.shareability = walkstate.mattrs.shareability;
else
    mattrs.shareability = EffectiveShareability(mattrs);

// Output Address
oa = StageOA(ZeroExtend(va, 64), walkparams.tgx, walkstate);
ipa = CreateAddressDescriptor(ZeroExtend(va, 64), oa, mattrs);

return (fault, ipa);
```

Library pseudocode for aarch32/translation/translation/AArch32.S1TranslateSD

```
// AArch32.S1TranslateSD()
// =======================
// Perform a stage 1 translation using short-descriptor format mapping VA to IPA/PA
// depending on the regime

(FaultRecord, AddressDescriptor, SDFType) AArch32.S1TranslateSD(FaultRecord fault_in, Regime regime,
                                                                SecurityState ss, bits(32) va,
                                                                AccType acctype, boolean aligned,
                                                                boolean iswrite, boolean ispriv)

    FaultRecord fault = fault_in;
    fault.secondstage = FALSE;
    fault.s2fs1walk   = FALSE;

    if !AArch32.S1Enabled(regime, ss) then
        AddressDescriptor ipa;
        (fault, ipa) = AArch32.S1DisabledOutput(fault, regime, ss, va, acctype, aligned);
        return (fault, ipa, SDFType UNKNOWN);

    TTWState walkstate;
    (fault, walkstate) = AArch32.S1WalkSD(fault, regime, ss, va, ispriv);

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN, SDFType UNKNOWN);

    domain = AArch32.OutputDomain(regime, walkstate.domain);
    SetInGuardedPage(FALSE);  // AArch32-VMSA does not guard any pages

    bit ntlsmd;
    if HaveTrapLoadStoreMultipleDeviceExt() then
        case regime of
            when Regime_EL30 ntlsmd = SCTLR_S.nTLSMD;
            when Regime_EL10 ntlsmd = if HaveAArch32EL(EL3) then SCTLR_NS.nTLSMD else SCTLR.nTLSMD;
    else
        ntlsmd = '1';

    if AArch32.S1HasAlignmentFault(acctype, aligned, ntlsmd, walkstate.mematTrs) then
        fault.statuscode = Fault_Alignment;
    elsif !(acctype IN {AccType_IC, AccType_DC}) && domain == Domain_NoAccess then
        fault.statuscode = Fault_Domain;
    elsif domain == Domain_Client then
        if IsAtomicRW(acctype) then
            if AArch32.S1SDHasPermissionsFault(regime, ss, walkstate.permissions,
                                               walkstate.mematTrs.memtype,
                                               walkstate.baseaddress.paspace,
                                               ispriv, acctype, FALSE) then
                // The Permission fault was not caused by lack of write permissions
                fault.statuscode = Fault_Permission;
                fault.write      = FALSE;
            elsif AArch32.S1SDHasPermissionsFault(regime, ss, walkstate.permissions,
                                                  walkstate.mematTrs.memtype,
                                                  walkstate.baseaddress.paspace,
                                                  ispriv, acctype, TRUE) then
                // The Permission fault was caused by lack of write permissions
                fault.statuscode = Fault_Permission;
                fault.write      = TRUE;
        elsif AArch32.S1SDHasPermissionsFault(regime, ss, walkstate.permissions,
                                              walkstate.mematTrs.memtype,
                                              walkstate.baseaddress.paspace,
                                              ispriv, acctype, iswrite) then
            fault.statuscode = Fault_Permission;

    if fault.statuscode != Fault_None then
        fault.domain = walkstate.domain;
        return (fault, AddressDescriptor UNKNOWN, walkstate.sdftype);

    MemoryAttributes mematTrs;
    if ((acctype == AccType_IFETCH &&
            (walkstate.mematTrs.memtype == MemType_Device || !AArch32.S1ICacheEnabled(regime))) ||
        (acctype != AccType_IFETCH &&
```

```
          walkstate.memattrs.memtype == MemType_Normal && !AArch32.S1DCacheEnabled(regime))) then
    // Treat memory attributes as Normal Non-Cacheable
    memattrs = NormalNCMemAttr();
    memattrs.xs = walkstate.memattrs.xs;
else
    memattrs = walkstate.memattrs;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime_EL10 && AArch32.EL2Enabled(ss) &&
        (if ELStateUsingAArch32(EL2, ss == SS_Secure) then HCR.VM else HCR_EL2.VM) == '1' &&
        !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
    memattrs.shareability = walkstate.memattrs.shareability;
else
    memattrs.shareability = EffectiveShareability(memattrs);

// Output Address
oa = AArch32.SDStageOA(walkstate.baseaddress, va, walkstate.sdftype);
ipa = CreateAddressDescriptor(ZeroExtend(va, 64), oa, memattrs);

return (fault, ipa, walkstate.sdftype);
```

Library pseudocode for aarch32/translation/translation/AArch32.S2Translate

```
// AArch32.S2Translate()
// =====================
// Perform a stage 2 translation mapping an IPA to a PA

(FaultRecord, AddressDescriptor) AArch32.S2Translate(FaultRecord fault_in, AddressDescriptor ipa,
                                                     SecurityState ss, boolean s2fs1walk,
                                                     AccType acctype, boolean aligned,
                                                     boolean iswrite, boolean ispriv)

    FaultRecord fault = fault_in;
    assert IsZero(ipa.paddress.address<51:40>);

    if !ELStateUsingAArch32(EL2, ss == SS_Secure) then
        s1aarch64 = FALSE;
        return AArch64.S2Translate(fault, ipa, s1aarch64, ss, s2fs1walk, acctype,
                                   aligned, iswrite, ispriv);

    // Prepare fault fields in case a fault is detected
    fault.statuscode  = Fault_None;
    fault.secondstage = TRUE;
    fault.s2fs1walk   = s2fs1walk;
    fault.ipaddress   = ipa.paddress;

    walkparams = AArch32.GetS2TTWParams();

    if walkparams.vm == '0' then
        // Stage 2 is disabled
        return (fault, ipa);

    if AArch32.IPAIsOutOfRange(walkparams, ipa.paddress.address<39:0>) then
        fault.statuscode = Fault_Translation;
        fault.level      = 1;
        return (fault, AddressDescriptor UNKNOWN);

    TTWState walkstate;
    (fault, walkstate) = AArch32.S2Walk(fault, walkparams, ipa);

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);

    if AArch32.S2HasAlignmentFault(acctype, aligned, walkstate.memattrs) then
        fault.statuscode = Fault_Alignment;
    elsif IsAtomicRW(acctype) then
        assert !s2fs1walk; // AArch32 does not support HW update of TT
        if AArch32.S2HasPermissionsFault(s2fs1walk, walkparams,
                                         walkstate.permissions,
                                         walkstate.memattrs.memtype,
                                         ispriv, acctype, FALSE) then
            // The Permission fault was not caused by lack of write permissions
            fault.statuscode = Fault_Permission;
            fault.write      = FALSE;
        elsif AArch32.S2HasPermissionsFault(s2fs1walk, walkparams,
                                            walkstate.permissions,
                                            walkstate.memattrs.memtype,
                                            ispriv, acctype, TRUE) then
            // The Permission fault was caused by lack of write permissions
            fault.statuscode = Fault_Permission;
            fault.write      = TRUE;
    elsif AArch32.S2HasPermissionsFault(s2fs1walk, walkparams,
                                        walkstate.permissions,
                                        walkstate.memattrs.memtype,
                                        ispriv, acctype, iswrite) then
        fault.statuscode = Fault_Permission;
    MemoryAttributes s2_memattrs;
    if ((s2fs1walk &&
             walkstate.memattrs.memtype == MemType_Device) ||
        (acctype == AccType_IFETCH &&
            (walkstate.memattrs.memtype == MemType_Device || HCR2.ID == '1')) ||
        (acctype != AccType_IFETCH &&
             walkstate.memattrs.memtype == MemType_Normal && HCR2.CD == '1')) then
```

```
        // Treat memory attributes as Normal Non-Cacheable
        s2_mematTrs = NormalNCMemAttr();
        s2_mematTrs.xs = walkstate.mematTrs.xs;
    else
        s2_mematTrs = walkstate.mematTrs;

    mematTrs = S2CombineS1MemAttrs(ipa.mematTrs, s2_mematTrs);
    ipa_64 = ZeroExtend(ipa.paddress.address<39:0>, 64);
    // Output Address
    oa = StageOA(ipa_64, walkparams.tgx, walkstate);
    pa = CreateAddressDescriptor(ipa.vaddress, oa, mematTrs);

    return (fault, pa);
```

Library pseudocode for aarch32/translation/translation/AArch32.SDStageOA

```
// AArch32.SDStageOA()
// ===================
// Given the final walk state of a short-descriptor translation walk,
// map the untranslated input address bits to the base output address

FullAddress AArch32.SDStageOA(FullAddress baseaddress, bits(32) va, SDFType sdftype)
    integer tsize;
    case sdftype of
        when SDFType_SmallPage      tsize = 12;
        when SDFType_LargePage      tsize = 16;
        when SDFType_Section        tsize = 20;
        when SDFType_Supersection   tsize = 24;

    // Output Address
    FullAddress oa;
    oa.address = baseaddress.address<51:tsize>:va<tsize-1:0>;
    oa.paspace = baseaddress.paspace;
    return oa;
```

Library pseudocode for aarch32/translation/translation/AArch32.TranslateAddress

```
// AArch32.TranslateAddress()
// ==========================
// Main entry point for translating an address

AddressDescriptor AArch32.TranslateAddress(bits(32) va, AccType acctype,
                                           boolean iswrite, boolean aligned,
                                           integer size)

    regime = TranslationRegime(PSTATE.EL, acctype);
    if !RegimeUsingAArch32(regime) then
        return AArch64.TranslateAddress(ZeroExtend(va, 64), acctype, iswrite,
                                        aligned, size);
    result = AArch32.FullTranslate(va, acctype, iswrite, aligned);
    if !IsFault(result) then
        result.fault = AArch32.CheckDebug(va, acctype, iswrite, size);

    // Update virtual address for abort functions
    result.vaddress = ZeroExtend(va, 64);

    return result;
```

Library pseudocode for aarch32/translation/walk/AArch32.DecodeDescriptorTypeLD

```
// AArch32.DecodeDescriptorTypeLD()
// ================================
// Determine whether the long-descriptor is a page, block or table

DescriptorType AArch32.DecodeDescriptorTypeLD(bits(64) descriptor, integer level)
    if descriptor<1:0> == '11' && level == FINAL_LEVEL then
        return DescriptorType_Page;
    elsif descriptor<1:0> == '11' then
        return DescriptorType_Table;
    elsif descriptor<1:0> == '01' && level != FINAL_LEVEL then
        return DescriptorType_Block;
    else
        return DescriptorType_Invalid;
```

Library pseudocode for aarch32/translation/walk/AArch32.DecodeDescriptorTypeSD

```
// AArch32.DecodeDescriptorTypeSD()
// ================================
// Determine the type of the short-descriptor

SDFType AArch32.DecodeDescriptorTypeSD(bits(32) descriptor, integer level)
    if level == 1 && descriptor<1:0> == '01' then
        return SDFType_Table;
    elsif level == 1 && descriptor<18,1> == '01' then
        return SDFType_Section;
    elsif level == 1 && descriptor<18,1> == '11' then
        return SDFType_Supersection;
    elsif level == 2 && descriptor<1:0> == '01' then
        return SDFType_LargePage;
    elsif level == 2 && descriptor<1:0> IN {'1x'} then
        return SDFType_SmallPage;
    else
        return SDFType_Invalid;
```

Library pseudocode for aarch32/translation/walk/AArch32.S1IASize

```
// AArch32.S1IASize()
// ==================
// Retrieve the number of bits containing the input address for stage 1 translation

integer AArch32.S1IASize(bits(3) txsz)
    return 32 - UInt(txsz);
```

Library pseudocode for aarch32/translation/walk/AArch32.S1WalkLD

```
// AArch32.S1WalkLD()
// ==================
// Traverse stage 1 translation tables in long format to obtain the final descriptor

(FaultRecord, TTWState) AArch32.S1WalkLD(FaultRecord fault_in, Regime regime, SecurityState ss,
                                         S1TTWParams walkparams, bits(32) va, boolean ispriv)
    FaultRecord fault = fault_in;
    bits(3) txsz;
    bits(64) ttbr;
    bit epd;
    if regime == Regime_EL2 then
        ttbr = HTTBR;
        txsz = walkparams.t0sz;
    else
        varange = AArch32.GetVARange(va, walkparams.t0sz, walkparams.t1sz);
        bits(64) ttbr0;
        bits(64) ttbr1;
        TTBCR_Type ttbcr;
        if regime == Regime_EL30 then
            ttbcr = TTBCR_S;
            ttbr0 = TTBR0_S;
            ttbr1 = TTBR1_S;
        elsif HaveAArch32EL(EL3) then
            ttbcr = TTBCR_NS;
            ttbr0 = TTBR0_NS;
            ttbr1 = TTBR1_NS;
        else
            ttbcr = TTBCR;
            ttbr0 = TTBR0;
            ttbr1 = TTBR1;

        assert ttbcr.EAE == '1';
        if varange == VARange_LOWER then
            txsz = walkparams.t0sz;
            ttbr = ttbr0;
            epd  = ttbcr.EPD0;
        else
            txsz = walkparams.t1sz;
            ttbr = ttbr1;
            epd  = ttbcr.EPD1;

    if regime != Regime_EL2 && epd == '1' then
        fault.level      = 1;
        fault.statuscode = Fault_Translation;
        return (fault, TTWState UNKNOWN);

    // Input Address size
    iasize      = AArch32.S1IASize(txsz);
    granulebits = TGxGranuleBits(walkparams.tgx);
    stride      = granulebits - 3;
    startlevel  = FINAL_LEVEL - (((iasize-1) - granulebits) DIV stride);
    levels      = FINAL_LEVEL - startlevel;

    if !IsZero(ttbr<47:40>) then
        fault.statuscode = Fault_AddressSize;
        fault.level      = 0;
        return (fault, TTWState UNKNOWN);

    FullAddress baseaddress;
    baselsb = (iasize - (levels*stride + granulebits)) + 3;
    baseaddress.paspace = if ss == SS_Secure then PAS_Secure else PAS_NonSecure;
    baseaddress.address = ZeroExtend(ttbr<39:baselsb>:Zeros(baselsb), 52);

    TTWState walkstate;
    walkstate.baseaddress = baseaddress;
    walkstate.level       = startlevel;
    walkstate.istable     = TRUE;
    // In regimes that support global and non-global translations, translation
    // table entries from lookup levels other than the final level of lookup
    // are treated as being non-global
```

```
walkstate.nG          = if HasUnprivileged(regime) then '1' else '0';
walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);
walkstate.permissions.ap_table  = '00';
walkstate.permissions.xn_table  = '0';
walkstate.permissions.pxn_table = '0';

indexmsb = iasize - 1;
bits(64) descriptor;
AddressDescriptor walkaddress;

walkaddress.vaddress = ZeroExtend(va, 64);

if !AArch32.S1DCacheEnabled(regime) then
    walkaddress.memattrs = NormalNCMemAttr();
    walkaddress.memattrs.xs = walkstate.memattrs.xs;
else
    walkaddress.memattrs = walkstate.memattrs;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime_EL10 && AArch32.EL2Enabled(ss) &&
    (if ELStateUsingAArch32(EL2, ss == SS_Secure) then HCR.VM else HCR_EL2.VM) == '1' &&
        !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
    walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
else
    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

integer indexlsb;
DescriptorType desctype;
repeat
    fault.level = walkstate.level;
    indexlsb = (FINAL_LEVEL - walkstate.level)*stride + granulebits;
    bits(40) index = ZeroExtend(va<indexmsb:indexlsb>:'000', 40);

    walkaddress.paddress.address = walkstate.baseaddress.address OR ZeroExtend(index, 52);
    walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

    // If there are two stages of translation, then the first stage table walk addresses
    // are themselves subject to translation
    if regime == Regime_EL10 && AArch32.EL2Enabled(ss) then
        s2fs1walk = TRUE;
        s2acctype = AccType_TTW;
        s2aligned = TRUE;
        s2write   = FALSE;
        (s2fault, s2walkaddress) = AArch32.S2Translate(fault, walkaddress, ss, s2fs1walk,
                                                       s2acctype, s2aligned, s2write, ispriv);
        // Check for a fault on the stage 2 walk
        if s2fault.statuscode != Fault_None then
            return (s2fault, TTWState UNKNOWN);

        (fault, descriptor) = FetchDescriptor(walkparams.ee, s2walkaddress, fault, 64);
    else
        (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, fault, 64);

    if fault.statuscode != Fault_None then
        return (fault, TTWState UNKNOWN);

    desctype = AArch32.DecodeDescriptorTypeLD(descriptor, walkstate.level);

    case desctype of
        when DescriptorType_Table
            if !IsZero(descriptor<47:40>) then
                fault.statuscode = Fault_AddressSize;
                return (fault, TTWState UNKNOWN);

            walkstate.baseaddress.address = ZeroExtend(descriptor<39:12>:Zeros(12), 52);
            if walkstate.baseaddress.paspace == PAS_Secure && descriptor<63> == '1' then
                walkstate.baseaddress.paspace = PAS_NonSecure;

            if walkparams.hpd == '0' then
```

```
                walkstate.permissions.xn_table  = (walkstate.permissions.xn_table  OR
                                                   descriptor<60>);
                walkstate.permissions.ap_table  = (walkstate.permissions.ap_table  OR
                                                   descriptor<62:61>);
                walkstate.permissions.pxn_table = (walkstate.permissions.pxn_table OR
                                                   descriptor<59>);

            walkstate.level = walkstate.level + 1;
            indexmsb = indexlsb - 1;

        when DescriptorType_Invalid
            fault.statuscode = Fault_Translation;
            return (fault, TTWState UNKNOWN);

        when DescriptorType_Page, DescriptorType_Block
            walkstate.istable = FALSE;

until desctype IN {DescriptorType_Page, DescriptorType_Block};

// Check the output address is inside the supported range
if !IsZero(descriptor<47:40>) then
    fault.statuscode = Fault_AddressSize;
    return (fault, TTWState UNKNOWN);

// Check the access flag
if descriptor<10> == '0' then
    fault.statuscode = Fault_AccessFlag;
    return (fault, TTWState UNKNOWN);

walkstate.permissions.xn  = descriptor<54>;
walkstate.permissions.pxn = descriptor<53>;
walkstate.permissions.ap  = descriptor<7:6>:'1';
walkstate.contiguous      = descriptor<52>;
if regime == Regime_EL2 then
    // All EL2 regime accesses are treated as Global
    walkstate.nG = '0';
elsif ss == SS_Secure && walkstate.baseaddress.paspace == PAS_NonSecure then
    // When a PE is using the Long-descriptor translation table format,
    // and is in Secure state, a translation must be treated as non-global,
    // regardless of the value of the nG bit,
    // if NSTable is set to 1 at any level of the translation table walk.
    walkstate.nG = '1';
else
    walkstate.nG = descriptor<11>;

walkstate.baseaddress.address = ZeroExtend(descriptor<39:indexlsb>:Zeros(indexlsb), 52);
if walkstate.baseaddress.paspace == PAS_Secure && descriptor<5> == '1' then
    walkstate.baseaddress.paspace = PAS_NonSecure;

mecattr = descriptor<4:2>;
sh      = descriptor<9:8>;
attr    = MAIRAttr(UInt(mecattr), walkparams.mair);
s1aarch64 = FALSE;
walkstate.mecattrs = S1DecodeMemAttrs(attr, sh, s1aarch64);

return (fault, walkstate);
```

Library pseudocode for aarch32/translation/walk/AArch32.S1WalkSD

```
// AArch32.S1WalkSD()
// ==================
// Traverse stage 1 translation tables in short format to obtain the final descriptor

(FaultRecord, TTWState) AArch32.S1WalkSD(FaultRecord fault_in, Regime regime, SecurityState ss,
                                         bits(32) va, boolean ispriv)
    FaultRecord fault = fault_in;
    SCTLR_Type sctlr;
    TTBCR_Type ttbcr;
    TTBR0_Type ttbr0;
    TTBR1_Type ttbr1;
    // Determine correct translation control registers to use.
    if regime == Regime_EL30 then
        sctlr = SCTLR_S;
        ttbcr = TTBCR_S;
        ttbr0 = TTBR0_S;
        ttbr1 = TTBR1_S;
    elsif HaveAArch32EL(EL3) then
        sctlr = SCTLR_NS;
        ttbcr = TTBCR_NS;
        ttbr0 = TTBR0_NS;
        ttbr1 = TTBR1_NS;
    else
        sctlr = SCTLR;
        ttbcr = TTBCR;
        ttbr0 = TTBR0;
        ttbr1 = TTBR1;

    assert ttbcr.EAE == '0';
    ee  = sctlr.EE;
    afe = sctlr.AFE;
    tre = sctlr.TRE;
    n = UInt(ttbcr.N);
    bits(32) ttb;
    bits(1) pd;
    bits(2) irgn;
    bits(2) rgn;
    bits(1) s;
    bits(1) nos;
    if n == 0 || IsZero(va<31:(32-n)>) then
        ttb  = ttbr0.TTB0:Zeros(7);
        pd   = ttbcr.PD0;
        irgn = ttbr0.IRGN;
        rgn  = ttbr0.RGN;
        s    = ttbr0.S;
        nos  = ttbr0.NOS;
    else
        n    = 0;  // TTBR1 translation always treats N as 0
        ttb  = ttbr1.TTB1:Zeros(7);
        pd   = ttbcr.PD1;
        irgn = ttbr1.IRGN;
        rgn  = ttbr1.RGN;
        s    = ttbr1.S;
        nos  = ttbr1.NOS;

    // Check if Translation table walk disabled for translations with this Base register.
    if pd == '1' then
        fault.level      = 1;
        fault.statuscode = Fault_Translation;
        return (fault, TTWState UNKNOWN);

    FullAddress baseaddress;
    baseaddress.paspace = if ss == SS_Secure then PAS_Secure else PAS_NonSecure;
    baseaddress.address = ZeroExtend(ttb<31:14-n>:Zeros(14-n), 52);

    TTWState walkstate;
    walkstate.baseaddress = baseaddress;
    // In regimes that support global and non-global translations, translation
    // table entries from lookup levels other than the final level of lookup
    // are treated as being non-global. Translations in Short-Descriptor Format
```

```
// always support global & non-global translations.
walkstate.nG          = '1';
walkstate.mematttrs   = WalkMemAttrs(s:nos, irgn, rgn);
walkstate.level       = 1;
walkstate.istable     = TRUE;

bits(4) domain;
bits(32) descriptor;
AddressDescriptor walkaddress;

walkaddress.vaddress = ZeroExtend(va, 64);

if !AArch32.S1DCacheEnabled(regime) then
    walkaddress.mematttrs = NormalNCMemAttr();
    walkaddress.mematttrs.xs = walkstate.mematttrs.xs;
else
    walkaddress.mematttrs = walkstate.mematttrs;

// Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
// to be either effective value or descriptor value
if (regime == Regime_EL10 && AArch32.EL2Enabled(ss) &&
    (if ELStateUsingAArch32(EL2, ss == SS_Secure) then HCR.VM else HCR_EL2.VM) == '1' &&
        !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
    walkaddress.mematttrs.shareability = walkstate.mematttrs.shareability;
else
    walkaddress.mematttrs.shareability = EffectiveShareability(walkaddress.mematttrs);

bit nG;
bit ns;
bit pxn;
bits(3) ap;
bits(3) tex;
bit c;
bit b;
bit xn;
repeat
    fault.level = walkstate.level;

    bits(32) index;
    if walkstate.level == 1 then
        index = ZeroExtend(va<31-n:20>:'00', 32);
    else
        index = ZeroExtend(va<19:12>:'00', 32);

    walkaddress.paddress.address = walkstate.baseaddress.address OR ZeroExtend(index,
                                                                                52);
    walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

    if regime == Regime_EL10 && AArch32.EL2Enabled(ss) then
        s2fs1walk = TRUE;
        s2acctype = AccType_TTW;
        s2aligned = TRUE;
        s2write   = FALSE;
        (s2fault, s2walkaddress) = AArch32.S2Translate(fault, walkaddress, ss, s2fs1walk,
                                                       s2acctype, s2aligned, s2write, ispriv);

        if s2fault.statuscode != Fault_None then
            return (s2fault, TTWState UNKNOWN);

        (fault, descriptor) = FetchDescriptor(ee, s2walkaddress, fault, 32);
    else
        (fault, descriptor) = FetchDescriptor(ee, walkaddress, fault, 32);

    if fault.statuscode != Fault_None then
        return (fault, TTWState UNKNOWN);

    walkstate.sdftype = AArch32.DecodeDescriptorTypeSD(descriptor, walkstate.level);

    case walkstate.sdftype of
        when SDFType_Invalid
```

```
            fault.domain     = domain;
            fault.statuscode = Fault_Translation;
            return (fault, TTWState UNKNOWN);

        when SDFType_Table
            domain = descriptor<8:5>;
            ns     = descriptor<3>;
            pxn    = descriptor<2>;

            walkstate.baseaddress.address = ZeroExtend(descriptor<31:10>:Zeros(10),
                                                       52);
            walkstate.level = 2;

        when SDFType_SmallPage
            nG  = descriptor<11>;
            s   = descriptor<10>;
            ap  = descriptor<9,5:4>;
            tex = descriptor<8:6>;
            c   = descriptor<3>;
            b   = descriptor<2>;
            xn  = descriptor<0>;

            walkstate.baseaddress.address = ZeroExtend(descriptor<31:12>:Zeros(12),
                                                       52);
            walkstate.istable = FALSE;

        when SDFType_LargePage
            xn  = descriptor<15>;
            tex = descriptor<14:12>;
            nG  = descriptor<11>;
            s   = descriptor<10>;
            ap  = descriptor<9,5:4>;
            c   = descriptor<3>;
            b   = descriptor<2>;

            walkstate.baseaddress.address = ZeroExtend(descriptor<31:16>:Zeros(16),
                                                       52);
            walkstate.istable = FALSE;

        when SDFType_Section
            ns     = descriptor<19>;
            nG     = descriptor<17>;
            s      = descriptor<16>;
            ap     = descriptor<15,11:10>;
            tex    = descriptor<14:12>;
            domain = descriptor<8:5>;
            xn     = descriptor<4>;
            c      = descriptor<3>;
            b      = descriptor<2>;
            pxn    = descriptor<0>;

            walkstate.baseaddress.address = ZeroExtend(descriptor<31:20>:Zeros(20),
                                                       52);
            walkstate.istable = FALSE;

        when SDFType_Supersection
            ns     = descriptor<19>;
            nG     = descriptor<17>;
            s      = descriptor<16>;
            ap     = descriptor<15,11:10>;
            tex    = descriptor<14:12>;
            xn     = descriptor<4>;
            c      = descriptor<3>;
            b      = descriptor<2>;
            pxn    = descriptor<0>;
            domain = '0000';

            walkstate.baseaddress.address = ZeroExtend(descriptor<8:5,23:20,31:24>:Zeros(24),
                                                       52);
            walkstate.istable = FALSE;
```

```
    until walkstate.sdftype != SDFType_Table;

    if afe == '1' && ap<0> == '0' then
        fault.domain     = domain;
        fault.statuscode = Fault_AccessFlag;
        return (fault, TTWState UNKNOWN);

    // Decode the TEX, C, B and S bits to produce target memory attributes
    if tre == '1' then
        walkstate.memattrs = AArch32.RemappedTEXDecode(regime, tex, c, b, s);
    elsif RemapRegsHaveResetValues() then
        walkstate.memattrs = AArch32.DefaultTEXDecode(tex, c, b, s);
    else
        walkstate.memattrs = MemoryAttributes IMPLEMENTATION_DEFINED;

    walkstate.permissions.ap  = ap;
    walkstate.permissions.xn  = xn;
    walkstate.permissions.pxn = pxn;
    walkstate.domain = domain;
    walkstate.nG     = nG;

    if ss == SS_Secure && ns == '0' then
        walkstate.baseaddress.paspace = PAS_Secure;
    else
        walkstate.baseaddress.paspace = PAS_NonSecure;

    return (fault, walkstate);
```

Library pseudocode for aarch32/translation/walk/AArch32.S2IASize

```
// AArch32.S2IASize()
// ==================
// Retrieve the number of bits containing the input address for stage 2 translation

integer AArch32.S2IASize(bits(4) t0sz)
    return 32 - SInt(t0sz);
```

Library pseudocode for aarch32/translation/walk/AArch32.S2StartLevel

```
// AArch32.S2StartLevel()
// ======================
// Determine the initial lookup level when performing a stage 2 translation
// table walk

integer AArch32.S2StartLevel(bits(2) sl0)
    return 2 - UInt(sl0);
```

Library pseudocode for aarch32/translation/walk/AArch32.S2Walk

```
// AArch32.S2Walk()
// ================
// Traverse stage 2 translation tables in long format to obtain the final descriptor

(FaultRecord, TTWState) AArch32.S2Walk(FaultRecord fault_in, S2TTWParams walkparams,
                                       AddressDescriptor ipa)
    FaultRecord fault = fault_in;

    if walkparams.sl0 IN {'1x'} || AArch32.S2InconsistentSL(walkparams) then
        fault.statuscode = Fault_Translation;
        fault.level      = 1;
        return (fault, TTWState UNKNOWN);

    // Input Address size
    iasize      = AArch32.S2IASize(walkparams.t0sz);
    startlevel  = AArch32.S2StartLevel(walkparams.sl0);
    levels      = FINAL_LEVEL - startlevel;
    granulebits = TGxGranuleBits(walkparams.tgx);
    stride      = granulebits - 3;

    if !IsZero(VTTBR<47:40>) then
        fault.statuscode = Fault_AddressSize;
        fault.level      = 0;
        return (fault, TTWState UNKNOWN);

    FullAddress baseaddress;
    baselsb = (iasize - (levels*stride + granulebits)) + 3;
    baseaddress.paspace = PAS_NonSecure;
    baseaddress.address = ZeroExtend(VTTBR<39:baselsb>:Zeros(baselsb), 52);

    TTWState walkstate;
    walkstate.baseaddress = baseaddress;
    walkstate.level       = startlevel;
    walkstate.istable     = TRUE;
    walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn,
                                         walkparams.orgn);

    indexmsb = iasize - 1;
    bits(64) descriptor;
    AddressDescriptor walkaddress;

    walkaddress.vaddress = ipa.vaddress;
    if HCR2.CD == '1' then
        walkaddress.memattrs = NormalNCMemAttr();
        walkaddress.memattrs.xs = walkstate.memattrs.xs;
    else
        walkaddress.memattrs = walkstate.memattrs;

    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

    integer indexlsb;
    DescriptorType desctype;
    repeat
        fault.level = walkstate.level;

        indexlsb = (FINAL_LEVEL - walkstate.level)*stride + granulebits;
        bits(40) index = ZeroExtend(ipa.paddress.address<indexmsb:indexlsb>:'000', 40);

        walkaddress.paddress.address = walkstate.baseaddress.address OR ZeroExtend(index, 52);
        walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

        (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, fault, 64);

        if fault.statuscode != Fault_None then
            return (fault, TTWState UNKNOWN);

        desctype = AArch32.DecodeDescriptorTypeLD(descriptor, walkstate.level);

        case desctype of
            when DescriptorType_Table
```

```
                if !IsZero(descriptor<47:40>) then
                    fault.statuscode = Fault_AddressSize;
                    return (fault, TTWState UNKNOWN);

                walkstate.baseaddress.address = ZeroExtend(descriptor<39:12>:Zeros(12), 52);
                walkstate.level = walkstate.level + 1;
                indexmsb = indexlsb - 1;

            when DescriptorType_Invalid
                fault.statuscode = Fault_Translation;
                return (fault, TTWState UNKNOWN);

            when DescriptorType_Page, DescriptorType_Block
                walkstate.istable = FALSE;

    until desctype IN {DescriptorType_Page, DescriptorType_Block};

    // Check the output address is inside the supported range
    if !IsZero(descriptor<47:40>) then
        fault.statuscode = Fault_AddressSize;
        return (fault, TTWState UNKNOWN);

    // Check the access flag
    if descriptor<10> == '0' then
        fault.statuscode = Fault_AccessFlag;
        return (fault, TTWState UNKNOWN);

    // Unpack the descriptor into address and upper and lower block attributes
    walkstate.baseaddress.address = ZeroExtend(descriptor<39:indexlsb>:Zeros(indexlsb), 52);

    walkstate.permissions.s2ap = descriptor<7:6>;
    walkstate.permissions.s2xn = descriptor<54>;
    if HaveExtendedExecuteNeverExt() then
        walkstate.permissions.s2xnx = descriptor<53>;
    else
        walkstate.permissions.s2xnx = '0';

    memattr = descriptor<5:2>;
    sh      = descriptor<9:8>;
    s2aarch64 = FALSE;
    walkstate.memattrs   = S2DecodeMemAttrs(memattr, sh, s2aarch64);
    walkstate.contiguous = descriptor<52>;

    return (fault, walkstate);
```

Library pseudocode for aarch32/translation/walk/AArch32.TranslationSizeSD

```
// AArch32.TranslationSizeSD()
// ===========================
// Determine the size of the translation

integer AArch32.TranslationSizeSD(SDFType sdftype)
    integer tsize;
    case sdftype of
        when SDFType_SmallPage      tsize = 12;
        when SDFType_LargePage      tsize = 16;
        when SDFType_Section        tsize = 20;
        when SDFType_Supersection   tsize = 24;

    return tsize;
```

Library pseudocode for aarch32/translation/walk/RemapRegsHaveResetValues

```
boolean RemapRegsHaveResetValues();
```

Library pseudocode for aarch32/translation/walkparams/AArch32.GetS1TTWParams

```
// AArch32.GetS1TTWParams()
// ========================
// Returns stage 1 translation table walk parameters from respective controlling
// System registers.

S1TTWParams AArch32.GetS1TTWParams(Regime regime, bits(32) va)
    S1TTWParams walkparams;

    case regime of
        when Regime_EL2  walkparams = AArch32.S1TTWParamsEL2();
        when Regime_EL10 walkparams = AArch32.S1TTWParamsEL10(va);
        when Regime_EL30 walkparams = AArch32.S1TTWParamsEL30(va);

    return walkparams;
```

Library pseudocode for aarch32/translation/walkparams/AArch32.GetS2TTWParams

```
// AArch32.GetS2TTWParams()
// ========================
// Gather walk parameters for stage 2 translation

S2TTWParams AArch32.GetS2TTWParams()
    S2TTWParams walkparams;

    walkparams.tgx  = TGx_4KB;
    walkparams.s    = VTCR.S;
    walkparams.t0sz = VTCR.T0SZ;
    walkparams.sl0  = VTCR.SL0;
    walkparams.irgn = VTCR.IRGN0;
    walkparams.orgn = VTCR.ORGN0;
    walkparams.sh   = VTCR.SH0;
    walkparams.ee   = HSCTLR.EE;
    walkparams.ptw  = HCR.PTW;
    walkparams.vm   = HCR.VM OR HCR.DC;

    // VTCR.S must match VTCR.T0SZ[3]
    if walkparams.s != walkparams.t0sz<3> then
        (-, walkparams.t0sz) = ConstrainUnpredictableBits(Unpredictable_RESVTCRS, 4);

    return walkparams;
```

Library pseudocode for aarch32/translation/walkparams/AArch32.GetVARange

```
// AArch32.GetVARange()
// ====================
// Select the translation base address for stage 1 long-descriptor walks

VARange AArch32.GetVARange(bits(32) va, bits(3) t0sz, bits(3) t1sz)
    // Lower range Input Address size
    lo_iasize = AArch32.S1IASize(t0sz);
    // Upper range Input Address size
    up_iasize = AArch32.S1IASize(t1sz);

    if t1sz == '000' && t0sz == '000' then
        return VARange_LOWER;
    elsif t1sz == '000' then
        return if IsZero(va<31:lo_iasize>) then VARange_LOWER else VARange_UPPER;
    elsif t0sz == '000' then
        return if IsOnes(va<31:up_iasize>) then VARange_UPPER else VARange_LOWER;
    elsif IsZero(va<31:lo_iasize>) then
        return VARange_LOWER;
    elsif IsOnes(va<31:up_iasize>) then
        return VARange_UPPER;
    else
        // Will be reported as a Translation Fault
        return VARange UNKNOWN;
```

Library pseudocode for aarch32/translation/walkparams/AArch32.S1DCacheEnabled

```
// AArch32.S1DCacheEnabled()
// =========================
// Determine cacheability of stage 1 data accesses

boolean AArch32.S1DCacheEnabled(Regime regime)
    case regime of
        when Regime_EL30 return SCTLR_S.C == '1';
        when Regime_EL2  return HSCTLR.C == '1';
        when Regime_EL10 return (if HaveAArch32EL(EL3) then SCTLR_NS.C else SCTLR.C) == '1';
```

Library pseudocode for aarch32/translation/walkparams/AArch32.S1ICacheEnabled

```
// AArch32.S1ICacheEnabled()
// =========================
// Determine cacheability of stage 1 instruction fetches

boolean AArch32.S1ICacheEnabled(Regime regime)
    case regime of
        when Regime_EL30 return SCTLR_S.I == '1';
        when Regime_EL2  return HSCTLR.I == '1';
        when Regime_EL10 return (if HaveAArch32EL(EL3) then SCTLR_NS.I else SCTLR.I) == '1';
```

Library pseudocode for aarch32/translation/walkparams/AArch32.S1TTWParamsEL10

```
// AArch32.S1TTWParamsEL10()
// =========================
// Gather stage 1 translation table walk parameters for EL1&0 regime
// (with EL2 enabled or disabled).

S1TTWParams AArch32.S1TTWParamsEL10(bits(32) va)
    bits(64) mair;
    bit sif;
    TTBCR_Type ttbcr;
    TTBCR2_Type ttbcr2;
    SCTLR_Type sctlr;

    if HaveAArch32EL(EL3) then
        ttbcr  = TTBCR_NS;
        ttbcr2 = TTBCR2_NS;
        sctlr  = SCTLR_NS;
        mair   = MAIR1_NS:MAIR0_NS;
        sif    = SCR.SIF;
    else
        ttbcr  = TTBCR;
        ttbcr2 = TTBCR2;
        sctlr  = SCTLR;
        mair   = MAIR1:MAIR0;
        sif    = SCR_EL3.SIF;

    assert ttbcr.EAE == '1';
    S1TTWParams walkparams;

    walkparams.t0sz   = ttbcr.T0SZ;
    walkparams.t1sz   = ttbcr.T1SZ;
    walkparams.ee     = sctlr.EE;
    walkparams.wxn    = sctlr.WXN;
    walkparams.uwxn   = sctlr.UWXN;
    walkparams.ntlsmd = if HaveTrapLoadStoreMultipleDeviceExt() then sctlr.nTLSMD else '1';
    walkparams.mair   = mair;
    walkparams.sif    = sif;

    varange = AArch32.GetVARange(va, walkparams.t0sz, walkparams.t1sz);
    if varange == VARange_LOWER then
        walkparams.sh   = ttbcr.SH0;
        walkparams.irgn = ttbcr.IRGN0;
        walkparams.orgn = ttbcr.ORGN0;
        walkparams.hpd  = if AArch32.HaveHPDExt() then ttbcr.T2E AND ttbcr2.HPD0 else '0';
    else
        walkparams.sh   = ttbcr.SH1;
        walkparams.irgn = ttbcr.IRGN1;
        walkparams.orgn = ttbcr.ORGN1;
        walkparams.hpd  = if AArch32.HaveHPDExt() then ttbcr.T2E AND ttbcr2.HPD1 else '0';

    return walkparams;
```

Library pseudocode for aarch32/translation/walkparams/AArch32.S1TTWParamsEL2

```
// AArch32.S1TTWParamsEL2()
// =======================
// Gather stage 1 translation table walk parameters for EL2 regime

S1TTWParams AArch32.S1TTWParamsEL2()
    S1TTWParams walkparams;

    walkparams.tgx  = TGx_4KB;
    walkparams.t0sz = HTCR.T0SZ;
    walkparams.irgn = HTCR.SH0;
    walkparams.orgn = HTCR.IRGN0;
    walkparams.sh   = HTCR.ORGN0;
    walkparams.hpd  = if AArch32.HaveHPDExt() then HTCR.HPD else '0';
    walkparams.ee   = HSCTLR.EE;
    walkparams.wxn  = HSCTLR.WXN;
    if HaveTrapLoadStoreMultipleDeviceExt() then
        walkparams.ntlsmd = HSCTLR.nTLSMD;
    else
        walkparams.ntlsmd = '1';

    walkparams.mair = HMAIR1:HMAIR0;

    return walkparams;
```

Library pseudocode for aarch32/translation/walkparams/AArch32.S1TTWParamsEL30

```
// AArch32.S1TTWParamsEL30()
// ========================
// Gather stage 1 translation table walk parameters for EL3&0 regime

S1TTWParams AArch32.S1TTWParamsEL30(bits(32) va)
    assert TTBCR_S.EAE == '1';
    S1TTWParams walkparams;

    walkparams.t0sz   = TTBCR_S.T0SZ;
    walkparams.t1sz   = TTBCR_S.T1SZ;
    walkparams.ee     = SCTLR_S.EE;
    walkparams.wxn    = SCTLR_S.WXN;
    walkparams.uwxn   = SCTLR_S.UWXN;
    walkparams.ntlsmd = if HaveTrapLoadStoreMultipleDeviceExt() then SCTLR_S.nTLSMD else '1';
    walkparams.mair   = MAIR1_S:MAIR0_S;
    walkparams.sif    = SCR.SIF;

    varange = AArch32.GetVARange(va, walkparams.t0sz, walkparams.t1sz);
    if varange == VARange_LOWER then
        walkparams.sh   = TTBCR_S.SH0;
        walkparams.irgn = TTBCR_S.IRGN0;
        walkparams.orgn = TTBCR_S.ORGN0;
        walkparams.hpd  = if AArch32.HaveHPDExt() then TTBCR_S.T2E AND TTBCR2_S.HPD0 else '0';
    else
        walkparams.sh   = TTBCR_S.SH1;
        walkparams.irgn = TTBCR_S.IRGN1;
        walkparams.orgn = TTBCR_S.ORGN1;
        walkparams.hpd  = if AArch32.HaveHPDExt() then TTBCR_S.T2E AND TTBCR2_S.HPD1 else '0';

    return walkparams;
```

Library pseudocode for aarch64/debug/brbe/BRBCycleCountingEnabled

```
// BRBCycleCountingEnabled()
// =========================
// Returns TRUE if the BRBINF<n>_EL1.{CCU, CC} fields are valid, FALSE otherwise.

boolean BRBCycleCountingEnabled()
    if EL2Enabled() && BRBCR_EL2.CC == '0' then return FALSE;
    if BRBCR_EL1.CC == '0' then return FALSE;
    return TRUE;
```

Library pseudocode for aarch64/debug/brbe/BRBEBranch

```
// BRBEBranch()
// ============
// Called to write branch record for the following branches when BRB is active:
// direct branches,
// indirect branches,
// direct branches with link,
// indirect branches with link,
// returns from subroutines.

BRBEBranch(BranchType br_type, boolean cond, bits(64) target_address)
    if BranchRecordAllowed(PSTATE.EL) && FilterBranchRecord(br_type, cond) then
        bits(6) branch_type;
        case br_type of
            when BranchType_DIR
                branch_type = if cond then '001000' else '000000';
            when BranchType_INDIR      branch_type = '000001';
            when BranchType_DIRCALL    branch_type = '000010';
            when BranchType_INDCALL    branch_type = '000011';
            when BranchType_RET        branch_type = '000101';
            otherwise                  Unreachable();

        bit ccu;
        bits(14) cc;
        (ccu, cc) = BranchEncCycleCount();
        bit lastfailed = if HaveTME() then BRBFCR_EL1.LASTFAILED else '0';
        bit transactional = if HaveTME() && TSTATE.depth > 0 then '1' else '0';
        bits(2) el = PSTATE.EL;
        bit mispredict = if BRBEMispredictAllowed() && BranchMispredict() then '1' else '0';

        UpdateBranchRecordBuffer(ccu, cc, lastfailed, transactional, branch_type, el, mispredict,
                                 '11', PC[], target_address);

        BRBFCR_EL1.LASTFAILED = '0';

        PMUEvent(PMU_EVENT_BRB_FILTRATE);

    return;
```

Library pseudocode for aarch64/debug/brbe/BRBEBranchOnISB

```
// BRBEBranchOnISB()
// =================
// Returns TRUE if ISBs generate Branch records, and FALSE otherwise.

boolean BRBEBranchOnISB()
    return boolean IMPLEMENTATION_DEFINED "ISB generates Branch records";
```

Library pseudocode for aarch64/debug/brbe/BRBEDebugStateExit

```
// BRBEDebugStateExit()
// ====================
// Called to write Debug state exit branch record when BRB is active.

BRBEDebugStateExit(bits(64) target_address)
    if BranchRecordAllowed(PSTATE.EL) then
        // Debug state is a prohibited region, therefore ccu=1, cc=0, source_address=0
        bits(6) branch_type = '111001';
        bit ccu = '1';
        bits(14) cc = Zeros(14);
        bit lastfailed = if HaveTME() then BRBFCR_EL1.LASTFAILED else '0';
        bit transactional = '0';
        bits(2) el = PSTATE.EL;
        bit mispredict = '0';

        UpdateBranchRecordBuffer(ccu, cc, lastfailed, transactional, branch_type, el, mispredict,
                                 '01', Zeros(64), target_address);

        BRBFCR_EL1.LASTFAILED = '0';

        PMUEvent(PMU_EVENT_BRB_FILTRATE);

    return;
```

Library pseudocode for aarch64/debug/brbe/BRBEException

```
// BRBEException()
// ===============
// Called to write exception branch record when BRB is active.

BRBEException(Exception exception, bits(64) preferred_exception_return,
              bits(64) target_address_in, bits(2) target_el, boolean trappedsyscallinst)
    bits(64) target_address = target_address_in;
    case target_el of
        when EL3  if !HaveBRBEv1p1() || (MDCR_EL3.E3BREC == MDCR_EL3.E3BREW) then return;
        when EL2  if BRBCR_EL2.EXCEPTION == '0' then return;
        when EL1  if BRBCR_EL1.EXCEPTION == '0' then return;

    boolean source_valid = BranchRecordAllowed(PSTATE.EL);
    boolean target_valid = BranchRecordAllowed(target_el);

    if source_valid || target_valid then
        bits(6) branch_type;
        case exception of
            when Exception_Uncategorized        branch_type = '100011'; // Trap
            when Exception_WFxTrap              branch_type = '100011'; // Trap
            when Exception_CP15RTTrap           branch_type = '100011'; // Trap
            when Exception_CP15RRTTrap          branch_type = '100011'; // Trap
            when Exception_CP14RTTrap           branch_type = '100011'; // Trap
            when Exception_CP14DTTrap           branch_type = '100011'; // Trap
            when Exception_AdvSIMDFPAccessTrap  branch_type = '100011'; // Trap
            when Exception_FPIDTrap             branch_type = '100011'; // Trap
            when Exception_PACTrap              branch_type = '100011'; // Trap
            when Exception_TSTARTAccessTrap     branch_type = '100011'; // Trap
            when Exception_CP14RRTTrap          branch_type = '100011'; // Trap
            when Exception_BranchTarget         branch_type = '101011'; // Inst Fault
            when Exception_IllegalState         branch_type = '100011'; // Trap
            when Exception_SupervisorCall
                if !trappedsyscallinst then     branch_type = '100010'; // Call
                else                            branch_type = '100011'; // Trap
            when Exception_HypervisorCall       branch_type = '100010'; // Call
            when Exception_MonitorCall
                if !trappedsyscallinst then     branch_type = '100010'; // Call
                else                            branch_type = '100011'; // Trap
            when Exception_SystemRegisterTrap   branch_type = '100011'; // Trap
            when Exception_SVEAccessTrap        branch_type = '100011'; // Trap
            when Exception_SMEAccessTrap        branch_type = '100011'; // Trap
            when Exception_ERetTrap             branch_type = '100011'; // Trap
            when Exception_PACFail              branch_type = '101100'; // Data Fault
            when Exception_InstructionAbort     branch_type = '101011'; // Inst Fault
            when Exception_PCAlignment          branch_type = '101010'; // Alignment
            when Exception_DataAbort            branch_type = '101100'; // Data Fault
            when Exception_NV2DataAbort         branch_type = '101100'; // Data Fault
            when Exception_SPAlignment          branch_type = '101010'; // Alignment
            when Exception_FPTrappedException   branch_type = '100011'; // Trap
            when Exception_SError               branch_type = '100100'; // System Error
            when Exception_Breakpoint           branch_type = '100110'; // Inst debug
            when Exception_SoftwareStep         branch_type = '100110'; // Inst debug
            when Exception_Watchpoint           branch_type = '100111'; // Data debug
            when Exception_NV2Watchpoint        branch_type = '100111'; // Data debug
            when Exception_SoftwareBreakpoint   branch_type = '100110'; // Inst debug
            when Exception_IRQ                  branch_type = '101110'; // IRQ
            when Exception_FIQ                  branch_type = '101111'; // FIQ
            when Exception_MemCpyMemSet         branch_type = '100011'; // Trap
            otherwise                           Unreachable();

        bit ccu;
        bits(14) cc;
        (ccu, cc) = BranchEncCycleCount();
        bit lastfailed = if HaveTME() then BRBFCR_EL1.LASTFAILED else '0';
        bit transactional = if source_valid && HaveTME() && TSTATE.depth > 0 then '1' else '0';
        bits(2) el = if target_valid then target_el else '00';
        bit mispredict = '0';
        bit sv = if source_valid then '1' else '0';
        bit tv = if target_valid then '1' else '0';
        bits(64) source_address = if source_valid then preferred_exception_return else Zeros(64);
```

```
        if !target_valid then
            target_address = Zeros(64);
        else
            target_address = AArch64.BranchAddr(target_address, target_el);

        UpdateBranchRecordBuffer(ccu, cc, lastfailed, transactional, branch_type, el, mispredict,
                                 sv:tv, source_address, target_address);

        BRBFCR_EL1.LASTFAILED = '0';

        PMUEvent(PMU_EVENT_BRB_FILTRATE);

    return;
```

Library pseudocode for aarch64/debug/brbe/BRBEExceptionReturn

```
// BRBEExceptionReturn()
// =====================
// Called to write exception return branch record when BRB is active.

BRBEExceptionReturn(bits(64) target_address_in, bits(2) source_el)
    bits(64) target_address = target_address_in;
    case source_el of
        when EL3  if !HaveBRBEv1p1() || (MDCR_EL3.E3BREC == MDCR_EL3.E3BREW) then return;
        when EL2  if BRBCR_EL2.ERTN == '0' then return;
        when EL1  if BRBCR_EL1.ERTN == '0' then return;

    boolean source_valid = BranchRecordAllowed(source_el);
    boolean target_valid = BranchRecordAllowed(PSTATE.EL);

    if source_valid || target_valid then
        bits(6) branch_type = '000111';
        bit ccu;
        bits(14) cc;
        (ccu, cc) = BranchEncCycleCount();
        bit lastfailed = if HaveTME() then BRBFCR_EL1.LASTFAILED else '0';
        bit transactional = if source_valid && HaveTME() && TSTATE.depth > 0 then '1' else '0';
        bits(2) el = if target_valid then PSTATE.EL else '00';
        bit mispredict = if source_valid && BRBEMispredictAllowed() && BranchMispredict() then '1' else
        bit sv = if source_valid then '1' else '0';
        bit tv = if target_valid then '1' else '0';
        bits(64) source_address = if source_valid then PC[] else Zeros(64);
        if !target_valid then
            target_address = Zeros(64);

        UpdateBranchRecordBuffer(ccu, cc, lastfailed, transactional, branch_type, el, mispredict,
                                 sv:tv, source_address, target_address);

        BRBFCR_EL1.LASTFAILED = '0';

        PMUEvent(PMU_EVENT_BRB_FILTRATE);

    return;
```

Library pseudocode for aarch64/debug/brbe/BRBEFreeze

```
// BRBEFreeze()
// ============
// Generates BRBE freeze event.

BRBEFreeze()
    BRBFCR_EL1.PAUSED = '1';
    BRBTS_EL1 = GetTimestamp(BRBETimeStamp());
```

Library pseudocode for aarch64/debug/brbe/BRBEISB

```
// BRBEISB()
// =========
// Handles ISB instruction for BRBE.

BRBEISB()
    boolean branch_conditional = FALSE;
    BRBEBranch(BranchType_DIR, branch_conditional, PC[] + 4);
```

Library pseudocode for aarch64/debug/brbe/BRBEMispredictAllowed

```
// BRBEMispredictAllowed()
// =======================
// Returns TRUE if the recording of branch misprediction is allowed, FALSE otherwise.

boolean BRBEMispredictAllowed()
    if EL2Enabled() && BRBCR_EL2.MPRED == '0' then return FALSE;
    if BRBCR_EL1.MPRED == '0' then return FALSE;
    return TRUE;
```

Library pseudocode for aarch64/debug/brbe/BRBETimeStamp

```
// BRBETimeStamp()
// ===============
// Returns captured timestamp.

TimeStamp BRBETimeStamp()
    if HaveEL(EL2) then
        TS_el2 = BRBCR_EL2.TS;
        if !HaveECVExt() && TS_el2 == '10' then
            // Reserved value
            (-, TS_el2) = ConstrainUnpredictableBits(Unpredictable_EL2TIMESTAMP, 2);
        case TS_el2 of
            when '00'
                // Falls out to check BRBCR_EL1.TS
            when '01'
                return TimeStamp_Virtual;
            when '10'
                assert HaveECVExt(); // Otherwise ConstrainUnpredictableBits removes this case
                return TimeStamp_OffsetPhysical;
            when '11'
                return TimeStamp_Physical;

    TS_el1 = BRBCR_EL1.TS;
    if TS_el1 == '00' || (!HaveECVExt() && TS_el1 == '10') then
        // Reserved value
        (-, TS_el1) = ConstrainUnpredictableBits(Unpredictable_EL1TIMESTAMP, 2);
    case TS_el1 of
        when '01'
            return TimeStamp_Virtual;
        when '10'
            return TimeStamp_OffsetPhysical;
        when '11'
            return TimeStamp_Physical;
        otherwise
            Unreachable();           // ConstrainUnpredictableBits removes this case
```

Library pseudocode for aarch64/debug/brbe/BRB_IALL

```
// BRB_IALL()
// ==========
// Called to perform invalidation of branch records

BRB_IALL()
    for i = 0 to GetBRBENumRecords() - 1
        Records_SRC[i] = Zeros(64);
        Records_TGT[i] = Zeros(64);
        Records_INF[i] = Zeros(64);
```

Library pseudocode for aarch64/debug/brbe/BRB_INJ

```
// BRB_INJ()
// =========
// Called to perform manual injection of branch records.

BRB_INJ()
    UpdateBranchRecordBuffer(BRBINFINJ_EL1.CCU, BRBINFINJ_EL1.CC, BRBINFINJ_EL1.LASTFAILED,
                             BRBINFINJ_EL1.T, BRBINFINJ_EL1.TYPE, BRBINFINJ_EL1.EL,
                             BRBINFINJ_EL1.MPRED, BRBINFINJ_EL1.VALID, BRBSRCINJ_EL1.ADDRESS,
                             BRBTGTINJ_EL1.ADDRESS);
    BRBINFINJ_EL1 = bits(64) UNKNOWN;
    BRBSRCINJ_EL1 = bits(64) UNKNOWN;
    BRBTGTINJ_EL1 = bits(64) UNKNOWN;

    if ConstrainUnpredictableBool(Unpredictable_BRBFILTRATE) then PMUEvent(PMU_EVENT_BRB_FILTRATE);
```

Library pseudocode for aarch64/debug/brbe/Branch

```
type BRBSRCType;
type BRBTGTType;
type BRBINFType;
```

Library pseudocode for aarch64/debug/brbe/BranchEncCycleCount

```
// The first return result is '1' if either of the following is true, and '0' otherwise:
// - This is the first Branch record after the PE exited a Prohibited Region.
// - This is the first Branch record after cycle counting has been enabled.
// If the first return return is '0', the second return result is the encoded cycle count
// since the last branch.
// The format of this field uses a mantissa and exponent to express the cycle count value.
//  - bits[7:0] indicate the mantissa M.
//  - bits[13:8] indicate the exponent E.
// The cycle count is expressed using the following function:
//   cycle_count = (if IsZero(E) then UInt(M) else UInt('1':M:Zeros(UInt(E)-1)))
// A value of all ones in both the mantissa and exponent indicates the cycle count value
// exceeded the size of the cycle counter.
// If the cycle count is not known, the second return result is zero.
(bit, bits(14)) BranchEncCycleCount();
```

Library pseudocode for aarch64/debug/brbe/BranchMispredict

```
// Returns TRUE if the branch being executed was mispredicted, FALSE otherwise.
boolean BranchMispredict();
```

Library pseudocode for aarch64/debug/brbe/BranchRawCycleCount

```
// If the cycle count is known, the return result is the cycle count since the last branch.
integer BranchRawCycleCount();
```

Library pseudocode for aarch64/debug/brbe/BranchRecordAllowed

```
// BranchRecordAllowed()
// =====================
// Returns TRUE if branch recording is allowed, FALSE otherwise.

boolean BranchRecordAllowed(bits(2) el)
    if ELUsingAArch32(el) then
        return FALSE;

    if BRBFCR_EL1.PAUSED == '1' then
        return FALSE;

    if el == EL3 && HaveBRBEv1p1() then
        return (MDCR_EL3.E3BREC != MDCR_EL3.E3BREW);

    if HaveEL(EL3) && (MDCR_EL3.SBRBE == '00' ||
        (CurrentSecurityState() == SS_Secure && MDCR_EL3.SBRBE == '01')) then
        return FALSE;

    case el of
        when EL3  return FALSE;                // FEAT_BRBEv1p1 not implemented
        when EL2  return BRBCR_EL2.E2BRE == '1';
        when EL1  return BRBCR_EL1.E1BRE == '1';
        when EL0
            if EL2Enabled() && HCR_EL2.TGE == '1' then
                return BRBCR_EL2.E0HBRE == '1';
            else
                return BRBCR_EL1.E0BRE == '1';
```

Library pseudocode for aarch64/debug/brbe/Contents

```
array [0..63] of BRBSRCType Records_SRC;

array [0..63] of BRBTGTType Records_TGT;

array [0..63] of BRBINFType Records_INF;
```

Library pseudocode for aarch64/debug/brbe/FilterBranchRecord

```
// FilterBranchRecord()
// ====================
// Returns TRUE if the branch record is not filtered out, FALSE otherwise.

boolean FilterBranchRecord(BranchType br, boolean cond)
    case br of
        when BranchType_DIRCALL
            return BRBFCR_EL1.DIRCALL != BRBFCR_EL1.EnI;
        when BranchType_INDCALL
            return BRBFCR_EL1.INDCALL != BRBFCR_EL1.EnI;
        when BranchType_RET
            return BRBFCR_EL1.RTN != BRBFCR_EL1.EnI;
        when BranchType_DIR
            if cond then
                return BRBFCR_EL1.CONDDIR != BRBFCR_EL1.EnI;
            else
                return BRBFCR_EL1.DIRECT != BRBFCR_EL1.EnI;
        when BranchType_INDIR
            return BRBFCR_EL1.INDIRECT != BRBFCR_EL1.EnI;
        otherwise  Unreachable();
    return FALSE;
```

Library pseudocode for aarch64/debug/brbe/FirstBranchAfterProhibited

```
// Returns TRUE if branch recorded is the first branch after a prohibited region,
// FALSE otherwise.
FirstBranchAfterProhibited();
```

Library pseudocode for aarch64/debug/brbe/GetBRBENumRecords

```
// GetBRBENumRecords()
// ===================
// Returns the number of branch records implemented.

integer GetBRBENumRecords()
    assert UInt(BRBIDR0_EL1.NUMREC) IN {0x08, 0x10, 0x20, 0x40};
    return integer IMPLEMENTATION_DEFINED "Number of BRB records";
```

Library pseudocode for aarch64/debug/brbe/Getter

```
// Getter functions for branch records
// ===================================
// Functions used by MRS instructions that access branch records

BRBSRCType BRBSRC_EL1[integer n]
    assert n IN {0..31};
    integer record = UInt(BRBFCR_EL1.BANK:n<4:0>);
    if record < GetBRBENumRecords() then
        return Records_SRC[record];
    else
        return Zeros(64);

BRBTGTType BRBTGT_EL1[integer n]
    assert n IN {0..31};
    integer record = UInt(BRBFCR_EL1.BANK:n<4:0>);
    if record < GetBRBENumRecords() then
        return Records_TGT[record];
    else
        return Zeros(64);

BRBINFType BRBINF_EL1[integer n]
    assert n IN {0..31};
    integer record = UInt(BRBFCR_EL1.BANK:n<4:0>);
    if record < GetBRBENumRecords() then
        return Records_INF[record];
    else
        return Zeros(64);
```

Library pseudocode for aarch64/debug/brbe/ShouldBRBEFreeze

```
// ShouldBRBEFreeze()
// ==================
// Returns TRUE if the BRBE freeze event conditions have been met, and FALSE otherwise.

boolean ShouldBRBEFreeze()
    if !BranchRecordAllowed(PSTATE.EL) then return FALSE;
    integer cntrs = GetNumEventCounters();
    if HaveEL(EL2) then
        integer hpmn = UInt(MDCR_EL2.HPMN);
        if BRBCR_EL1.FZP == '1' && hpmn != 0 && !IsZero(PMOVSCLR_EL0<hpmn-1:0>) then
            return TRUE;
        if BRBCR_EL2.FZP == '1' && cntrs != hpmn && !IsZero(PMOVSCLR_EL0<cntrs-1:hpmn>) then
            return TRUE;
    else
        if BRBCR_EL1.FZP == '1' && !IsZero(PMOVSCLR_EL0<cntrs-1:0>) then
            return TRUE;
    return FALSE;
```

Library pseudocode for aarch64/debug/brbe/UpdateBranchRecordBuffer

```
// UpdateBranchRecordBuffer()
// ==========================
// Add a new Branch record to the buffer.

UpdateBranchRecordBuffer(bit ccu, bits(14) cc, bit lastfailed, bit transactional, bits(6) branch_type, bi
                         bit mispredict, bits(2) valid, bits(64) source_address, bits(64) target_address)
    // Shift the Branch Records in the buffer
    for i = GetBRBENumRecords() - 1 downto 1
        Records_SRC[i] = Records_SRC[i - 1];
        Records_TGT[i] = Records_TGT[i - 1];
        Records_INF[i] = Records_INF[i - 1];

    Records_INF[0].CCU        = ccu;
    Records_INF[0].CC         = cc;

    Records_INF[0].EL         = el;
    Records_INF[0].VALID      = valid;
    Records_INF[0].T          = transactional;
    Records_INF[0].LASTFAILED = lastfailed;
    Records_INF[0].MPRED      = mispredict;
    Records_INF[0].TYPE       = branch_type;

    Records_SRC[0] = source_address;
    Records_TGT[0] = target_address;

    return;
```

Library pseudocode for aarch64/debug/breakpoint/AArch64.BreakpointMatch

```
// AArch64.BreakpointMatch()
// =========================
// Breakpoint matching in an AArch64 translation regime.

boolean AArch64.BreakpointMatch(integer n, bits(64) vaddress,
                                AccType acctype, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert n < NumBreakpointsImplemented();

    enabled = DBGBCR_EL1[n].E == '1';
    ispriv = PSTATE.EL != EL0;
    linked = DBGBCR_EL1[n].BT IN {'0x01'};
    isbreakpnt = TRUE;
    linked_to = FALSE;

    ssce = if HaveRME() then DBGBCR_EL1[n].SSCE else '0';
    state_match = AArch64.StateMatch(DBGBCR_EL1[n].SSC, ssce, DBGBCR_EL1[n].HMC, DBGBCR_EL1[n].PMC,
                                     linked, DBGBCR_EL1[n].LBN, isbreakpnt, acctype, ispriv);
    value_match = AArch64.BreakpointValueMatch(n, vaddress, linked_to);

    if HaveAArch32() && size == 4 then                    // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
        // event.
        match_i = AArch64.BreakpointValueMatch(n, vaddress + 2, linked_to);
        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);
    if vaddress<1> == '1' && DBGBCR_EL1[n].BAS == '1111' then
        // The above notwithstanding, if DBGBCR_EL1[n].BAS == '1111', then it is CONSTRAINED
        // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
        // at the address DBGBVR_EL1[n]+2.
        if value_match then value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);

    match = value_match && state_match && enabled;

    return match;
```

Library pseudocode for aarch64/debug/breakpoint/AArch64.BreakpointValueMatch

```
// AArch64.BreakpointValueMatch()
// ==============================

boolean AArch64.BreakpointValueMatch(integer n_in, bits(64) vaddress, boolean linked_to)

    // "n" is the identity of the breakpoint unit to match against.
    // "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
    //   matching breakpoints.
    // "linked_to" is TRUE if this is a call from StateMatch for linking.
    integer n = n_in;

    // If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
    // no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
    if n >= NumBreakpointsImplemented() then
        Constraint c;
        (c, n) = ConstrainUnpredictableInteger(0, NumBreakpointsImplemented() - 1, Unpredictable_BPNOTIMP
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return FALSE;

    // If this breakpoint is not enabled, it cannot generate a match. (This could also happen on a
    // call from StateMatch for linking).
    if DBGBCR_EL1[n].E == '0' then return FALSE;

    context_aware = (n >= (NumBreakpointsImplemented() - NumContextAwareBreakpointsImplemented()));

    // If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
    dbgtype = DBGBCR_EL1[n].BT;
    Constraint c;

    if ((dbgtype IN {'011x','11xx'} && !HaveVirtHostExt() && !HaveV82Debug()) ||    // Context matching
          dbgtype IN {'010x'} ||                                                     // Reserved
          (!(dbgtype IN {'0x0x'}) && !context_aware) ||                                 // Context match
          (dbgtype IN {'1xxx'} && !HaveEL(EL2))) then                                // EL2 extension
        (c, dbgtype) = ConstrainUnpredictableBits(Unpredictable_RESBPTYPE, 4);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return FALSE;
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    // Determine what to compare against.
    match_addr = (dbgtype IN {'0x0x'});
    match_vmid = (dbgtype IN {'10xx'});
    match_cid  = (dbgtype IN {'001x'});
    match_cid1 = (dbgtype IN { '101x', 'x11x'});
    match_cid2 = (dbgtype IN {'11xx'});
    linked     = (dbgtype IN {'xxx1'});

    // If this is a call from StateMatch, return FALSE if the breakpoint is not programmed for a
    // VMID and/or context ID match, of if not context-aware. The above assertions mean that the
    // code can just test for match_addr == TRUE to confirm all these things.
    if linked_to && (!linked || match_addr) then return FALSE;

    // If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
    if !linked_to && linked && !match_addr then return FALSE;

    // Do the comparison.
    boolean BVR_match;
    if match_addr then
        boolean byte_select_match;
        byte = UInt(vaddress<1:0>);
        if HaveAArch32() then
            // T32 instructions can be executed at EL0 in an AArch64 translation regime.
            assert byte IN {0,2};                 // "vaddress" is halfword aligned
            byte_select_match = (DBGBCR_EL1[n].BAS<byte> == '1');
        else
            assert byte == 0;                     // "vaddress" is word aligned
            byte_select_match = TRUE;             // DBGBCR_EL1[n].BAS<byte> is RES1
        // If the DBGxVR<n>_EL1.RESS field bits are not a sign extension of the MSB
        // of DBGBVR<n>_EL1.VA, it is UNPREDICTABLE whether they appear to be
        // included in the match.
        // If 'vaddress' is outside of the current virtual address space, then the access
```

```
        // generates a Translation fault.
        integer top = AArch64.VAMax();
        if !IsOnes(DBGBVR_EL1[n]<63:top>) && !IsZero(DBGBVR_EL1[n]<63:top>) then
            if ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS) then
                top = 63;
        BVR_match = (vaddress<top:2> == DBGBVR_EL1[n]<top:2>) && byte_select_match;

    elsif match_cid then
        if IsInHost() then
            BVR_match = (CONTEXTIDR_EL2<31:0> == DBGBVR_EL1[n]<31:0>);
        else
            BVR_match = (PSTATE.EL IN {EL0, EL1} && CONTEXTIDR_EL1<31:0> == DBGBVR_EL1[n]<31:0>);
    elsif match_cid1 then
        BVR_match = (PSTATE.EL IN {EL0, EL1} && !IsInHost() && CONTEXTIDR_EL1<31:0> == DBGBVR_EL1[n]<31:0
    boolean BXVR_match;
    if match_vmid then
        bits(16) vmid;
        bits(16) bvr_vmid;
        if !Have16bitVMID() || VTCR_EL2.VS == '0' then
            vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
            bvr_vmid = ZeroExtend(DBGBVR_EL1[n]<39:32>, 16);
        else
            vmid = VTTBR_EL2.VMID;
            bvr_vmid = DBGBVR_EL1[n]<47:32>;
        BXVR_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                      !IsInHost() &&
                      vmid == bvr_vmid);
    elsif match_cid2 then
        BXVR_match = (PSTATE.EL != EL3 && (HaveVirtHostExt() || HaveV82Debug()) &&
                      EL2Enabled() &&
                      DBGBVR_EL1[n]<63:32> == CONTEXTIDR_EL2<31:0>);

    bvr_match_valid = (match_addr || match_cid || match_cid1);
    bxvr_match_valid = (match_vmid || match_cid2);

    match = (!bxvr_match_valid || BXVR_match) && (!bvr_match_valid || BVR_match);

    return match;
```

Library pseudocode for aarch64/debug/breakpoint/AArch64.StateMatch

```
// AArch64.StateMatch()
// ====================
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch64.StateMatch(bits(2) SSC_in, bit SSCE_in, bit HMC_in,
                           bits(2) PxC_in, boolean linked_in, bits(4) LBN,
                           boolean isbreakpnt, AccType acctype, boolean ispriv)
    if !HaveRME() then assert SSCE_in == '0';

    // "SSC_in","SSCE_in","HMC_in","PxC_in" are the control fields from the DBGBCR[n] or DBGWCR[n] regist
    // "linked_in" is TRUE if this is a linked breakpoint/watchpoint type.
    // "LBN" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
    // "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.
    // "ispriv" is valid for watchpoints, and selects between privileged and unprivileged accesses.
    bits(2) SSC = SSC_in;
    bit SSCE = SSCE_in;
    bit HMC = HMC_in;
    bits(2) PxC = PxC_in;
    boolean linked = linked_in;

    // If parameters are set to a reserved type, behaves as either disabled or a defined type
    Constraint c;
    (c, SSC, SSCE, HMC, PxC) = CheckValidStateMatch(SSC, SSCE, HMC, PxC, isbreakpnt);
    if c == Constraint_DISABLED then return FALSE;
    // Otherwise the HMC,SSC,SSCE,PxC values are either valid or the values returned by
    // CheckValidStateMatch are valid.

    EL3_match = HaveEL(EL3) && HMC == '1' && SSC<0> == '0';
    EL2_match = HaveEL(EL2) && ((HMC == '1' && (SSC:PxC != '1000')) || SSC == '11');
    EL1_match = PxC<0> == '1';
    EL0_match = PxC<1> == '1';

    boolean priv_match;
    if HaveNV2Ext() && acctype == AccType_NV2REGISTER && !isbreakpnt then
        priv_match = EL2_match;
    elsif !ispriv && !isbreakpnt then
        priv_match = EL0_match;
    else
        case PSTATE.EL of
            when EL3  priv_match = EL3_match;
            when EL2  priv_match = EL2_match;
            when EL1  priv_match = EL1_match;
            when EL0  priv_match = EL0_match;

    boolean security_state_match;
    ss = CurrentSecurityState();
    case SSCE:SSC of
        when '000' security_state_match = HMC == '1' || ss != SS_Root;
        when '001' security_state_match = ss == SS_NonSecure;
        when '010' security_state_match = (HMC == '1' && ss == SS_Root) || ss == SS_Secure;
        when '011' security_state_match = (HMC == '1' && ss != SS_Root) || ss == SS_Secure;
        when '101' security_state_match = ss == SS_Realm;
    integer lbn;
    if linked then
        // "LBN" must be an enabled context-aware breakpoint unit. If it is not context-aware then
        // it is CONSTRAINED UNPREDICTABLE whether this gives no match, or LBN is mapped to some
        // UNKNOWN breakpoint that is context-aware.
        lbn = UInt(LBN);
        first_ctx_cmp = NumBreakpointsImplemented() - NumContextAwareBreakpointsImplemented();
        last_ctx_cmp = NumBreakpointsImplemented() - 1;
        if (lbn < first_ctx_cmp || lbn > last_ctx_cmp) then
            (c, lbn) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp, Unpredictable_BPNOTCTXC
            assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
            case c of
                when Constraint_DISABLED  return FALSE;      // Disabled
                when Constraint_NONE      linked = FALSE;    // No linking
                // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint

    boolean linked_match;
    if linked then
```

```
        vaddress = bits(64) UNKNOWN;
        linked_to = TRUE;
        linked_match = AArch64.BreakpointValueMatch(lbn, vaddress, linked_to);

    return priv_match && security_state_match && (!linked || linked_match);
```

Library pseudocode for aarch64/debug/enables/AArch64.GenerateDebugExceptions

```
// AArch64.GenerateDebugExceptions()
// =================================

boolean AArch64.GenerateDebugExceptions()
    ss = CurrentSecurityState();
    return AArch64.GenerateDebugExceptionsFrom(PSTATE.EL, ss, PSTATE.D);
```

Library pseudocode for aarch64/debug/enables/AArch64.GenerateDebugExceptionsFrom

```
// AArch64.GenerateDebugExceptionsFrom()
// =====================================

boolean AArch64.GenerateDebugExceptionsFrom(bits(2) from_el, SecurityState from_state, bit mask)

    if OSLSR_EL1.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;

    route_to_el2 = (HaveEL(EL2) && (from_state != SS_Secure || IsSecureEL2Enabled()) &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));
    target = (if route_to_el2 then EL2 else EL1);
    boolean enabled;
    if HaveEL(EL3) && from_state == SS_Secure then
        enabled = MDCR_EL3.SDD == '0';
        if from_el == EL0 && ELUsingAArch32(EL1) then
            enabled = enabled || SDER32_EL3.SUIDEN == '1';
    else
        enabled = TRUE;

    if from_el == target then
        enabled = enabled && MDSCR_EL1.KDE == '1' && mask == '0';
    else
        enabled = enabled && UInt(target) > UInt(from_el);

    return enabled;
```

Library pseudocode for aarch64/debug/pmu/AArch64.CheckForPMUOverflow

```
// AArch64.CheckForPMUOverflow()
// =============================
// Signal Performance Monitors overflow IRQ and CTI overflow events

AArch64.CheckForPMUOverflow()
    boolean pmuirq;
    bit E;
    if ShouldBRBEFreeze() then
        BRBEFreeze();
    pmuirq = PMCR_EL0.E == '1' && PMINTENSET_EL1.C == '1' && PMOVSSET_EL0.C == '1';
    integer counters = GetNumEventCounters();
    if counters != 0 then
        for idx = 0 to counters - 1
            E = if AArch64.PMUCounterIsHyp(idx) then MDCR_EL2.HPME else PMCR_EL0.E;
            if E == '1' && PMINTENSET_EL1<idx> == '1' && PMOVSSET_EL0<idx> == '1' then pmuirq = TRUE;

    SetInterruptRequestLevel(InterruptID_PMUIRQ, if pmuirq then HIGH else LOW);

    CTI_SetEventLevel(CrossTriggerIn_PMUOverflow, if pmuirq then HIGH else LOW);

    // The request remains set until the condition is cleared. (For example, an interrupt handler
    // or cross-triggered event handler clears the overflow status flag by writing to PMOVSCLR_EL0.)
```

Library pseudocode for aarch64/debug/pmu/AArch64.ClearEventCounters

```
// AArch64.ClearEventCounters()
// ============================
// Zero all the event counters.

AArch64.ClearEventCounters()
    integer counters = AArch64.GetNumEventCountersAccessible();
    if counters != 0 then
        for idx = 0 to counters - 1
            PMEVCNTR_EL0[idx] = Zeros(64);
```

Library pseudocode for aarch64/debug/pmu/AArch64.CountPMUEvents

```
// AArch64.CountPMUEvents()
// ========================
// Return TRUE if counter "idx" should count its event.
// For the cycle counter, idx == CYCLE_COUNTER_ID.

boolean AArch64.CountPMUEvents(integer idx)
    constant integer num_counters = GetNumEventCounters();
    assert idx == CYCLE_COUNTER_ID || idx < num_counters;
    boolean debug;
    boolean enabled;
    boolean prohibited;
    boolean filtered;
    boolean frozen;
    boolean resvd_for_el2;
    bit E;
    bit spme;
    bits(32) ovflws;
    // Event counting is disabled in Debug state
    debug = Halted();

    // Software can reserve some counters for EL2
    resvd_for_el2 = AArch64.PMUCounterIsHyp(idx);
    ss = CurrentSecurityState();

    // Main enable controls
    if idx == CYCLE_COUNTER_ID then
        enabled = PMCR_EL0.E == '1' && PMCNTENSET_EL0.C == '1';
    else
        E = if resvd_for_el2 then MDCR_EL2.HPME else PMCR_EL0.E;
        enabled = E == '1' && PMCNTENSET_EL0<idx> == '1';

    // Event counting is allowed unless it is prohibited by any rule below
    prohibited = FALSE;

    // Event counting in Secure state is prohibited if all of:
    // * EL3 is implemented
    // * MDCR_EL3.SPME == 0, and either:
    //   - FEAT_PMUv3p7 is not implemented
    //   - MDCR_EL3.MPMX == 0
    if HaveEL(EL3) && ss == SS_Secure then
        if HavePMUv3p7() then
            prohibited = MDCR_EL3.<SPME,MPMX> == '00';
        else
            prohibited = MDCR_EL3.SPME == '0';

    // Event counting at EL3 is prohibited if all of:
    // * FEAT_PMUv3p7 is implemented
    // * One of the following is true:
    //   - MDCR_EL3.SPME == 0
    //   - PMNx is not reserved for EL2
    // * MDCR_EL3.MPMX == 1
    if !prohibited && PSTATE.EL == EL3 && HavePMUv3p7() then
        prohibited = MDCR_EL3.MPMX == '1' && (MDCR_EL3.SPME == '0' || !resvd_for_el2);

    // Event counting at EL2 is prohibited if all of:
    // * The HPMD Extension is implemented
    // * PMNx is not reserved for EL2
    // * MDCR_EL2.HPMD == 1
    if !prohibited && PSTATE.EL == EL2 && HaveHPMDExt() && !resvd_for_el2 then
        prohibited = MDCR_EL2.HPMD == '1';

    // The IMPLEMENTATION DEFINED authentication interface might override software
    if prohibited && !HaveNoSecurePMUDisableOverride() then
        prohibited = !ExternalSecureNoninvasiveDebugEnabled();

    // Event counting might be frozen
    frozen = FALSE;

    // If FEAT_PMUv3p7 is implemented, event counting can be frozen
    if HavePMUv3p7() then
```

```
        hpmn = UInt(MDCR_EL2.HPMN);
        ovflws = ZeroExtend(PMOVSSET_EL0<num_counters-1:0>, 32);
        bit FZ;
        if resvd_for_el2 then
            FZ = MDCR_EL2.HPMFZO;
            ovflws<hpmn-1:0> = Zeros(hpmn);
        else
            FZ = PMCR_EL0.FZO;
            if HaveEL(EL2) && hpmn < num_counters then
                ovflws<num_counters-1:hpmn> = Zeros(num_counters - hpmn);
        frozen = (FZ == '1') && !IsZero(ovflws);

    // PMCR_EL0.DP disables the cycle counter when event counting is prohibited
    if (prohibited || frozen) && idx == CYCLE_COUNTER_ID then
        enabled = enabled && (PMCR_EL0.DP == '0');
        // Otherwise whether event counting is prohibited does not affect the cycle counter
        prohibited = FALSE;
        frozen = FALSE;

    // If FEAT_PMUv3p5 is implemented, cycle counting can be prohibited.
    // This is not overridden by PMCR_EL0.DP.
    if HavePMUv3p5() && idx == CYCLE_COUNTER_ID then
        if (HaveEL(EL3) && ss == SS_Secure &&
            MDCR_EL3.SCCD == '1') then
            prohibited = TRUE;
        if PSTATE.EL == EL2 && MDCR_EL2.HCCD == '1' then
            prohibited = TRUE;

    // If FEAT_PMUv3p7 is implemented, cycle counting an be prohibited at EL3.
    // This is not overriden by PMCR_EL0.DP.
    if HavePMUv3p7() && idx == CYCLE_COUNTER_ID then
        if PSTATE.EL == EL3 && MDCR_EL3.MCCD == '1' then
            prohibited = TRUE;

    // Event counting can be filtered by the {P, U, NSK, NSU, NSH, M, SH, RLK, RLU, RLH} bits
    filter = if idx == CYCLE_COUNTER_ID then PMCCFILTR_EL0<31:0> else PMEVTYPER_EL0[idx]<31:0>;

    P   = filter<31>;
    U   = filter<30>;
    NSK = if HaveEL(EL3) then filter<29> else '0';
    NSU = if HaveEL(EL3) then filter<28> else '0';
    NSH = if HaveEL(EL2) then filter<27> else '0';
    M   = if HaveEL(EL3) then filter<26> else '0';
    SH  = if HaveEL(EL3) && HaveSecureEL2Ext() then filter<24> else '0';
    RLK = if HaveRME() then filter<22> else '0';
    RLU = if HaveRME() then filter<21> else '0';
    RLH = if HaveRME() then filter<20> else '0';

    ss = CurrentSecurityState();
    case PSTATE.EL of
        when EL0
            case ss of
                when SS_NonSecure filtered = U != NSU;
                when SS_Secure    filtered = U == '1';
                when SS_Realm     filtered = U != RLU;
        when EL1
            case ss of
                when SS_NonSecure filtered = P != NSK;
                when SS_Secure    filtered = P == '1';
                when SS_Realm     filtered = P != RLK;
        when EL2
            case ss of
                when SS_NonSecure filtered = NSH == '0';
                when SS_Secure    filtered = NSH == SH;
                when SS_Realm     filtered = NSH == RLH;
        when EL3
            filtered = M != P;

    return !debug && enabled && !prohibited && !filtered && !frozen;
```

Library pseudocode for aarch64/debug/pmu/AArch64.GetNumEventCountersAccessible

```
// AArch64.GetNumEventCountersAccessible()
// =======================================
// Return the number of event counters that can be accessed at the current Exception level.

integer AArch64.GetNumEventCountersAccessible()
    integer n;
    integer total_counters = GetNumEventCounters();
    // Software can reserve some counters for EL2
    if PSTATE.EL IN {EL1, EL0} && EL2Enabled() then
        n = UInt(MDCR_EL2.HPMN);
        if n > total_counters || (!HaveFeatHPMN0() && n == 0) then
            (-, n) = ConstrainUnpredictableInteger(0, total_counters,
                                                   Unpredictable_PMUEVENTCOUNTER);
    else
        n = total_counters;

    return n;
```

Library pseudocode for aarch64/debug/pmu/AArch64.IncrementEventCounter

```
// AArch64.IncrementEventCounter()
// ===============================
// Increment the specified event counter by the specified amount.

AArch64.IncrementEventCounter(integer idx, integer increment)
    integer old_value;
    integer new_value;
    integer ovflw;
    bit lp;
    old_value = UInt(PMEVCNTR_EL0[idx]);
    new_value = old_value + PMUCountValue(idx, increment);

    if HavePMUv3p5() then
        PMEVCNTR_EL0[idx] = new_value<63:0>;
        lp = if AArch64.PMUCounterIsHyp(idx) then MDCR_EL2.HLP else PMCR_EL0.LP;
        ovflw = if lp == '1' then 64 else 32;
    else
        PMEVCNTR_EL0[idx] = ZeroExtend(new_value<31:0>, 64);
        ovflw = 32;

    if old_value<64:ovflw> != new_value<64:ovflw> then
        PMOVSSET_EL0<idx> = '1';
        PMOVSCLR_EL0<idx> = '1';
        // Check for the CHAIN event from an even counter
        if idx<0> == '0' && idx + 1 < GetNumEventCounters() && (!HavePMUv3p5() || lp == '0') then
            PMUEvent(PMU_EVENT_CHAIN, 1, idx + 1);
```

Library pseudocode for aarch64/debug/pmu/AArch64.PMUCounterIsHyp

```
// AArch64.PMUCounterIsHyp
// =======================
// Returns TRUE if a counter is reserved for use by EL2, FALSE otherwise.

boolean AArch64.PMUCounterIsHyp(integer n)
    boolean resvd_for_el2;
    // Software can reserve some event counters for EL2
    if n != CYCLE_COUNTER_ID && HaveEL(EL2) then
        resvd_for_el2 = n >= UInt(MDCR_EL2.HPMN);
        if UInt(MDCR_EL2.HPMN) > GetNumEventCounters() || (!HaveFeatHPMN0() && IsZero(MDCR_EL2.HPMN)) the
            resvd_for_el2 = boolean UNKNOWN;
    else
        resvd_for_el2 = FALSE;

    return resvd_for_el2;
```

Library pseudocode for aarch64/debug/pmu/AArch64.PMUCycle

```
// AArch64.PMUCycle()
// ==================
// Called at the end of each cycle to increment event counters and
// check for PMU overflow. In pseudocode, a cycle ends after the
// execution of the operational pseudocode.

AArch64.PMUCycle()
    if !HavePMUv3() then
        return;

    PMUEvent(PMU_EVENT_CPU_CYCLES);

    integer counters = GetNumEventCounters();
    if counters != 0 then
        for idx = 0 to counters - 1
            if AArch64.CountPMUEvents(idx) then
                accumulated = PMUEventAccumulator[idx];
                AArch64.IncrementEventCounter(idx, accumulated);
            PMUEventAccumulator[idx] = 0;

    integer old_value;
    integer new_value;
    integer ovflw;
    if (AArch64.CountPMUEvents(CYCLE_COUNTER_ID) &&
        (!HaveAArch32() || PMCR_EL0.LC == '1' || PMCR_EL0.D == '0' || HasElapsed64Cycles())) then
        old_value = UInt(PMCCNTR_EL0);
        new_value = old_value + 1;
        PMCCNTR_EL0 = new_value<63:0>;

        if HaveAArch32() then
            ovflw = if PMCR_EL0.LC == '1' then 64 else 32;
        else
            ovflw = 64;

        if old_value<64:ovflw> != new_value<64:ovflw> then
            PMOVSSET_EL0.C = '1';
            PMOVSCLR_EL0.C = '1';

    AArch64.CheckForPMUOverflow();
```

Library pseudocode for aarch64/debug/pmu/AArch64.PMUSwIncrement

```
// AArch64.PMUSwIncrement()
// ========================
// Generate PMU Events on a write to PMSWINC_EL0.

AArch64.PMUSwIncrement(bits(32) sw_incr)
    integer counters = AArch64.GetNumEventCountersAccessible();
    if counters != 0 then
        for idx = 0 to counters - 1
            if sw_incr<idx> == '1' then
                PMUEvent(PMU_EVENT_SW_INCR, 1, idx);
```

Library pseudocode for aarch64/debug/statisticalprofiling/CollectContextIDR1

```
// CollectContextIDR1()
// ====================

boolean CollectContextIDR1()
    if !StatisticalProfilingEnabled() then return FALSE;
    if  PSTATE.EL == EL2 then return FALSE;
    if EL2Enabled() && HCR_EL2.TGE == '1' then return FALSE;
    return PMSCR_EL1.CX == '1';
```

Library pseudocode for aarch64/debug/statisticalprofiling/CollectContextIDR2

```
// CollectContextIDR2()
// ====================

boolean CollectContextIDR2()
    if !StatisticalProfilingEnabled() then return FALSE;
    if !EL2Enabled() then return FALSE;
    return PMSCR_EL2.CX == '1';
```

Library pseudocode for aarch64/debug/statisticalprofiling/CollectPhysicalAddress

```
// CollectPhysicalAddress()
// ========================

boolean CollectPhysicalAddress()
    if !StatisticalProfilingEnabled() then return FALSE;
    (owning_ss, owning_el) = ProfilingBufferOwner();
    if HaveEL(EL2) && (owning_ss != SS_Secure || IsSecureEL2Enabled()) then
        return PMSCR_EL2.PA == '1' && (owning_el == EL2 || PMSCR_EL1.PA == '1');
    else
        return PMSCR_EL1.PA == '1';
```

Library pseudocode for aarch64/debug/statisticalprofiling/CollectTimeStamp

```
// CollectTimeStamp()
// ==================

TimeStamp CollectTimeStamp()
    if !StatisticalProfilingEnabled() then return TimeStamp_None;
    (-, owning_el) = ProfilingBufferOwner();

    if owning_el == EL2 then
        if PMSCR_EL2.TS == '0' then return TimeStamp_None;
    else
        if PMSCR_EL1.TS == '0' then return TimeStamp_None;

    bits(2) PCT_el1;
    if !HaveECVExt() then
        PCT_el1 = '0':PMSCR_EL1.PCT<0>;       // PCT<1> is RES0
    else
        PCT_el1 = PMSCR_EL1.PCT;
        if PCT_el1 == '10' then
            // Reserved value
            (-, PCT_el1) = ConstrainUnpredictableBits(Unpredictable_PMSCR_PCT, 2);
    if EL2Enabled() then
        bits(2) PCT_el2;
        if !HaveECVExt() then
            PCT_el2 = '0':PMSCR_EL2.PCT<0>;   // PCT<1> is RES0
        else
            PCT_el2 = PMSCR_EL2.PCT;
            if PCT_el2 == '10' then
                // Reserved value
                (-, PCT_el2) = ConstrainUnpredictableBits(Unpredictable_PMSCR_PCT, 2);
        case PCT_el2 of
            when '00'
                return if IsInHost() then TimeStamp_Physical else TimeStamp_Virtual;
            when '01'
                if owning_el == EL2 then return TimeStamp_Physical;
            when '11'
                assert HaveECVExt();          // FEAT_ECV must be implemented
                if owning_el == EL1 && PCT_el1 == '00' then
                    return if IsInHost() then TimeStamp_Physical else TimeStamp_Virtual;
                else
                    return TimeStamp_OffsetPhysical;
            otherwise
                Unreachable();

    case PCT_el1 of
        when '00' return if IsInHost() then TimeStamp_Physical else TimeStamp_Virtual;
        when '01' return TimeStamp_Physical;
        when '11'
            assert HaveECVExt();              // FEAT_ECV must be implemented
            return TimeStamp_OffsetPhysical;
        otherwise Unreachable();
```

Library pseudocode for aarch64/debug/statisticalprofiling/OpType

```
enumeration OpType {
    OpType_Load,           // Any memory-read operation other than atomics, compare-and-swap, and swap
    OpType_Store,          // Any memory-write operation, including atomics without return
    OpType_LoadAtomic,     // Atomics with return, compare-and-swap and swap
    OpType_Branch,         // Software write to the PC
    OpType_Other           // Any other class of operation
};
```

Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingBufferEnabled

```
// ProfilingBufferEnabled()
// ========================

boolean ProfilingBufferEnabled()
    if !HaveStatisticalProfiling() then return FALSE;
    (owning_ss, owning_el) = ProfilingBufferOwner();
    bits(2) state_bits;
    if HaveRME() then
        state_bits = SCR_EL3.<NSE,NS>;
    else
        state_bits = '0' : SCR_EL3.NS;

    boolean state_match;
    case owning_ss of
        when SS_Secure    state_match = state_bits == '00';
        when SS_NonSecure state_match = state_bits == '01';
        when SS_Realm     state_match = state_bits == '11';
    return (!ELUsingAArch32(owning_el) && state_match &&
            PMBLIMITR_EL1.E == '1' && PMBSR_EL1.S == '0');
```

Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingBufferOwner

```
// ProfilingBufferOwner()
// ======================

(SecurityState, bits(2)) ProfilingBufferOwner()
    SecurityState owning_ss;

    if HaveEL(EL3) then
        bits(3) state_bits;
        if HaveRME() then
            state_bits = MDCR_EL3.<NSPBE,NSPB>;
            if state_bits IN {'10x'} then
                // Reserved value
                (-, state_bits) = ConstrainUnpredictableBits(Unpredictable_RESERVEDNSxB, 3);
        else
            state_bits = '0' : MDCR_EL3.NSPB;

        case state_bits of
            when '00x' owning_ss = SS_Secure;
            when '01x' owning_ss = SS_NonSecure;
            when '11x' owning_ss = SS_Realm;
    else
        owning_ss = if SecureOnlyImplementation() then SS_Secure else SS_NonSecure;

    bits(2) owning_el;
    if HaveEL(EL2) && (owning_ss != SS_Secure || IsSecureEL2Enabled()) then
        owning_el = if MDCR_EL2.E2PB == '00' then EL2 else EL1;
    else
        owning_el = EL1;

    return (owning_ss, owning_el);
```

Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingSynchronizationBarrier

```
// Barrier to ensure that all existing profiling data has been formatted, and profiling buffer
// addresses have been translated such that writes to the profiling buffer have been initiated.
// A following DSB completes when writes to the profiling buffer have completed.
ProfilingSynchronizationBarrier();
```

Library pseudocode for aarch64/debug/statisticalprofiling/StatisticalProfilingEnabled

```
// StatisticalProfilingEnabled()
// =============================

boolean StatisticalProfilingEnabled()
    if !HaveStatisticalProfiling() || UsingAArch32() || !ProfilingBufferEnabled() then
        return FALSE;

    tge_set = EL2Enabled() && HCR_EL2.TGE == '1';
    (owning_ss, owning_el) = ProfilingBufferOwner();
    if (UInt(owning_el) <  UInt(PSTATE.EL) || (tge_set && owning_el == EL1) ||
        owning_ss != CurrentSecurityState())  then
        return FALSE;
    bit spe_bit;
    case PSTATE.EL of
        when EL3  Unreachable();
        when EL2  spe_bit = PMSCR_EL2.E2SPE;
        when EL1  spe_bit = PMSCR_EL1.E1SPE;
        when EL0  spe_bit = (if tge_set then PMSCR_EL2.E0HSPE else PMSCR_EL1.E0SPE);

    return spe_bit == '1';
```

Library pseudocode for aarch64/debug/statisticalprofiling/TimeStamp

```
enumeration TimeStamp {
    TimeStamp_None,              // No timestamp
    TimeStamp_CoreSight,         // CoreSight time (IMPLEMENTATION DEFINED)
    TimeStamp_Physical,          // Physical counter value with no offset
    TimeStamp_OffsetPhysical,    // Physical counter value minus CNTPOFF_EL2
    TimeStamp_Virtual  };        // Physical counter value minus CNTVOFF_EL2
```

Library pseudocode for aarch64/debug/takeexceptiondbg/AArch64.TakeExceptionInDebugState

```
// AArch64.TakeExceptionInDebugState()
// ===================================
// Take an exception in Debug state to an Exception level using AArch64.

AArch64.TakeExceptionInDebugState(bits(2) target_el, ExceptionRecord exception_in)
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);
    ExceptionRecord exception = exception_in;
    boolean sync_errors;
    boolean iesb_req;
    if HaveIESB() then
        sync_errors = SCTLR[target_el].IESB == '1';
        if HaveDoubleFaultExt() then
            sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' && target_el == EL3);
        // SCTLR[].IESB and/or SCR_EL3.NMEA (if applicable) might be ignored in Debug state.
        if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
            sync_errors = FALSE;
    else
        sync_errors = FALSE;

    if HaveTME() && TSTATE.depth > 0 then
        TMFailure cause;
        case exception.exceptype of
            when Exception_SoftwareBreakpoint cause = TMFailure_DBG;
            when Exception_Breakpoint         cause = TMFailure_DBG;
            when Exception_Watchpoint         cause = TMFailure_DBG;
            when Exception_SoftwareStep       cause = TMFailure_DBG;
            otherwise                         cause = TMFailure_ERR;
        FailTransaction(cause, FALSE);

    SynchronizeContext();

    // If coming from AArch32 state, the top parts of the X[] registers might be set to zero
    from_32 = UsingAArch32();
    if from_32 then AArch64.MaybeZeroRegisterUppers();
    if from_32 && HaveSME() && PSTATE.SM == '1' then
        ResetSVEState();
    else
        MaybeZeroSVEUppers(target_el);

    AArch64.ReportException(exception, target_el);

    PSTATE.EL = target_el;
    PSTATE.nRW = '0';
    PSTATE.SP = '1';

    SPSR[] = bits(64) UNKNOWN;
    ELR[] = bits(64) UNKNOWN;

    // PSTATE.{SS,D,A,I,F} are not observable and ignored in Debug state, so behave as if UNKNOWN.
    PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
    PSTATE.IL = '0';
    if from_32 then                             // Coming from AArch32
        PSTATE.IT = '00000000';
        PSTATE.T = '0';                         // PSTATE.J is RES0
    if (HavePANExt() && (PSTATE.EL == EL1 || (PSTATE.EL == EL2 && ELIsInHost(EL0))) &&
        SCTLR[].SPAN == '0') then
        PSTATE.PAN = '1';
    if HaveUAOExt() then PSTATE.UAO = '0';
    if HaveBTIExt() then PSTATE.BTYPE = '00';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    if HaveMTEExt() then PSTATE.TCO = '1';

    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(64) UNKNOWN;

    EDSCR.ERR = '1';
    UpdateEDSCRFields();                        // Update EDSCR processor state flags.

    if sync_errors then
        SynchronizeErrors();
```

```
    EndOfInstruction();
```

Library pseudocode for aarch64/debug/watchpoint/AArch64.WatchpointByteMatch

```
// AArch64.WatchpointByteMatch()
// =============================

boolean AArch64.WatchpointByteMatch(integer n, AccType acctype, bits(64) vaddress)

    integer top = AArch64.VAMax();
    bottom = if DBGWVR_EL1[n]<2> == '1' then 2 else 3;           // Word or doubleword
    byte_select_match = (DBGWCR_EL1[n].BAS<UInt(vaddress<bottom-1:0>)> != '0');
    mask = UInt(DBGWCR_EL1[n].MASK);

    // If DBGWCR_EL1[n].MASK is non-zero value and DBGWCR_EL1[n].BAS is not set to '11111111', or
    // DBGWCR_EL1[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
    // UNPREDICTABLE.
    if mask > 0 && !IsOnes(DBGWCR_EL1[n].BAS) then
        byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPMASKANDBAS);
    else
        LSB = (DBGWCR_EL1[n].BAS AND NOT(DBGWCR_EL1[n].BAS - 1));  MSB = (DBGWCR_EL1[n].BAS + LSB);
        if !IsZero(MSB AND (MSB - 1)) then                    // Not contiguous
            byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPBASCONTIGUOUS);
            bottom = 3;                                       // For the whole doubleword

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        Constraint c;
        (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable_RESWPMASK);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
        case c of
            when Constraint_DISABLED  return FALSE;           // Disabled
            when Constraint_NONE      mask = 0;               // No masking
            // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

    boolean WVR_match;
    if mask > bottom then
        // If the DBGxVR<n>_EL1.RESS field bits are not a sign extension of the MSB
        // of DBGBVR<n>_EL1.VA, it is UNPREDICTABLE whether they appear to be
        // included in the match.
        if !IsOnes(DBGBVR_EL1[n]<63:top>) && !IsZero(DBGBVR_EL1[n]<63:top>) then
            if ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS) then
                top = 63;
        WVR_match = (vaddress<top:mask> == DBGWVR_EL1[n]<top:mask>);
        // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR_EL1[n]<mask-1:bottom>) then
            WVR_match = ConstrainUnpredictableBool(Unpredictable_WPMASKEDBITS);
    else
        WVR_match = vaddress<top:bottom> == DBGWVR_EL1[n]<top:bottom>;

    return WVR_match && byte_select_match;
```

Library pseudocode for aarch64/debug/watchpoint/AArch64.WatchpointMatch

```
// AArch64.WatchpointMatch()
// =========================
// Watchpoint matching in an AArch64 translation regime.

boolean AArch64.WatchpointMatch(integer n, bits(64) vaddress, integer size, boolean ispriv,
                                AccType acctype, boolean iswrite)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert n < NumWatchpointsImplemented();

    // "ispriv" is:
    //  * FALSE for all loads, stores, and atomic operations executed at EL0.
    //  * FALSE if the access is unprivileged.
    //  * TRUE for all other loads, stores, and atomic operations.

    enabled = DBGWCR_EL1[n].E == '1';
    linked = DBGWCR_EL1[n].WT == '1';
    isbreakpnt = FALSE;

    ssce = if HaveRME() then DBGWCR_EL1[n].SSCE else '0';
    state_match = AArch64.StateMatch(DBGWCR_EL1[n].SSC, ssce, DBGWCR_EL1[n].HMC, DBGWCR_EL1[n].PAC,
                                     linked, DBGWCR_EL1[n].LBN, isbreakpnt, acctype, ispriv);
    ls_match = FALSE;
    if IsAtomicRW(acctype) then
        ls_match = (DBGWCR_EL1[n].LSC != '00');
    elsif iswrite then
        ls_match = (DBGWCR_EL1[n].LSC<1> != '0');
    else
        ls_match = (DBGWCR_EL1[n].LSC<0> != '0');

    value_match = FALSE;
    for byte = 0 to size - 1
        value_match = value_match || AArch64.WatchpointByteMatch(n, acctype, vaddress + byte);

    return value_match && state_match && ls_match && enabled;
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.Abort

```
// AArch64.Abort()
// ===============
// Abort and Debug exception handling in an AArch64 translation regime.

AArch64.Abort(bits(64) vaddress, FaultRecord fault)

    if IsDebugException(fault) then
        if fault.acctype == AccType_IFETCH then
            if UsingAArch32() && fault.debugmoe == DebugException_VectorCatch then
                AArch64.VectorCatchException(fault);
            else
                AArch64.BreakpointException(fault);
        else
            AArch64.WatchpointException(vaddress, fault);
    elsif fault.gpcf.gpf != GPCF_None && ReportAsGPCException(fault) then
        TakeGPCException(vaddress, fault);
    elsif fault.acctype == AccType_IFETCH then
        AArch64.InstructionAbort(vaddress, fault);
    else
        AArch64.DataAbort(vaddress, fault);
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.AbortSyndrome

```
// AArch64.AbortSyndrome()
// =======================
// Creates an exception syndrome record for Abort and Watchpoint exceptions
// from an AArch64 translation regime.

ExceptionRecord AArch64.AbortSyndrome(Exception exceptype, FaultRecord fault, bits(64) vaddress)
    exception = ExceptionSyndrome(exceptype);

    d_side = exceptype IN {Exception_DataAbort, Exception_NV2DataAbort, Exception_Watchpoint, Exception_N

    (exception.syndrome, exception.syndrome2) = AArch64.FaultSyndrome(d_side, fault);
    exception.vaddress = ZeroExtend(vaddress, 64);
    if IPAValid(fault) then
        exception.ipavalid = TRUE;
        exception.NS = if fault.ipaddress.paspace == PAS_NonSecure then '1' else '0';
        exception.ipaddress = fault.ipaddress.address;
    else
        exception.ipavalid = FALSE;

    return exception;
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.CheckPCAlignment

```
// AArch64.CheckPCAlignment()
// ==========================

AArch64.CheckPCAlignment()

    bits(64) pc = ThisInstrAddr(64);
    if pc<1:0> != '00' then
        AArch64.PCAlignmentFault();
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.DataAbort

```
// AArch64.DataAbort()
// ===================

AArch64.DataAbort(bits(64) vaddress, FaultRecord fault)
    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1' && IsExternalAbort(fault);
    route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
                    (HCR_EL2.TGE == '1' ||
                     (HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault)) ||
                     (HaveRME() && fault.gpcf.gpf == GPCF_Fail && HCR_EL2.GPF == '1') ||
                     (HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER) ||
                     IsSecondStage(fault)));

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    integer vect_offset;
    if (HaveDoubleFaultExt() && (PSTATE.EL == EL3 || route_to_el3) &&
        IsExternalAbort(fault) && SCR_EL3.EASE == '1') then
        vect_offset = 0x180;
    else
        vect_offset = 0x0;
    ExceptionRecord exception;
    if HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER then
        exception = AArch64.AbortSyndrome(Exception_NV2DataAbort, fault, vaddress);
    else
        exception = AArch64.AbortSyndrome(Exception_DataAbort, fault, vaddress);
    bits(2) target_el = EL1;
    if PSTATE.EL == EL3 || route_to_el3 then
        target_el = EL3;
    elsif PSTATE.EL == EL2 || route_to_el2 then
        target_el = EL2;
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.EffectiveTCF

```
// AArch64.EffectiveTCF()
// ======================
// Returns the TCF field applied to tag check faults in the given Exception level.

bits(2) AArch64.EffectiveTCF(AccType acctype)
    bits(2) tcf, el;
    el = S1TranslationRegime();

    if el == EL3 then
        tcf = SCTLR_EL3.TCF;
    elsif el == EL2 then
        if AArch64.AccessUsesEL(acctype) == EL0 then
            tcf = SCTLR_EL2.TCF0;
        else
            tcf = SCTLR_EL2.TCF;
    elsif el == EL1 then
        if AArch64.AccessUsesEL(acctype) == EL0 then
            tcf = SCTLR_EL1.TCF0;
        else
            tcf = SCTLR_EL1.TCF;

    if tcf == '11' then        //reserved value
        if !HaveMTE3Ext() then
            (-,tcf) = ConstrainUnpredictableBits(Unpredictable_RESTCF, 2);

    return tcf;
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.InstructionAbort

```
// AArch64.InstructionAbort()
// ==========================

AArch64.InstructionAbort(bits(64) vaddress, FaultRecord fault)
    // External aborts on instruction fetch must be taken synchronously
    if HaveDoubleFaultExt() then assert fault.statuscode != Fault_AsyncExternal;
    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1' && IsExternalAbort(fault);
    route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
                    (HCR_EL2.TGE == '1' ||
                     (HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault)) ||
                     (HaveRME() && fault.gpcf.gpf == GPCF_Fail && HCR_EL2.GPF == '1') ||
                     IsSecondStage(fault)));

    ExceptionRecord exception;
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    integer vect_offset;
    if (HaveDoubleFaultExt() && (PSTATE.EL == EL3 || route_to_el3) &&
        IsExternalAbort(fault) && SCR_EL3.EASE == '1') then
        vect_offset = 0x180;
    else
        vect_offset = 0x0;

    exception = AArch64.AbortSyndrome(Exception_InstructionAbort, fault, vaddress);

    bits(2) target_el = EL1;
    if PSTATE.EL == EL3 || route_to_el3 then
        target_el = EL3;
    elsif PSTATE.EL == EL2 || route_to_el2 then
        target_el = EL2;
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.PCAlignmentFault

```
// AArch64.PCAlignmentFault()
// ==========================
// Called on unaligned program counter in AArch64 state.

AArch64.PCAlignmentFault()

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_PCAlignment);
    exception.vaddress = ThisInstrAddr(64);
    bits(2) target_el = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.RaiseTagCheckFault

```
// AArch64.RaiseTagCheckFault()
// ============================
// Raise a tag check fault exception.

AArch64.RaiseTagCheckFault(bits(64) va, boolean write)
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    integer vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_DataAbort);
    exception.syndrome<5:0> = '010001';
    if write then
        exception.syndrome<6> = '1';

    exception.vaddress = bits(4) UNKNOWN : va<59:0>;

    bits(2) target_el = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.ReportTagCheckFault

```
// AArch64.ReportTagCheckFault()
// =============================
// Records a tag check fault exception into the appropriate TFSR_ELx.

AArch64.ReportTagCheckFault(bits(2) el, bit ttbr)
    if el == EL3 then
        assert ttbr == '0';
        TFSR_EL3.TF0 = '1';
    elsif el == EL2 then
        if ttbr == '0' then
            TFSR_EL2.TF0 = '1';
        else
            TFSR_EL2.TF1 = '1';
    elsif el == EL1 then
        if ttbr == '0' then
            TFSR_EL1.TF0 = '1';
        else
            TFSR_EL1.TF1 = '1';
    elsif el == EL0 then
        if ttbr == '0' then
            TFSRE0_EL1.TF0 = '1';
        else
            TFSRE0_EL1.TF1 = '1';
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.SPAlignmentFault

```
// AArch64.SPAlignmentFault()
// ==========================
// Called on an unaligned stack pointer in AArch64 state.

AArch64.SPAlignmentFault()

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SPAlignment);

    bits(2) target_el = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.TagCheckFault

```
// AArch64.TagCheckFault()
// =======================
// Handle a tag check fault condition.

AArch64.TagCheckFault(bits(64) vaddress, AccType acctype, boolean iswrite)
    bits(2) tcf, el;
    el  = AArch64.AccessUsesEL(acctype);
    tcf = AArch64.EffectiveTCF(acctype);
    case tcf of
        when '00'       // Tag Check Faults have no effect on the PE
            return;
        when '01'       // Tag Check Faults cause a synchronous exception
            AArch64.RaiseTagCheckFault(vaddress, iswrite);
        when '10'       // Tag Check Faults are asynchronously accumulated
            AArch64.ReportTagCheckFault(el, vaddress<55>);
        when '11'       // Tag Check Faults cause a synchronous exception on reads or on
                        // a read/write access, and are asynchronously accumulated on writes
            // Check for access performing both a read and a write.

            if !iswrite || IsAtomicRW(acctype) then
                AArch64.RaiseTagCheckFault(vaddress, iswrite);
            else
                AArch64.ReportTagCheckFault(el, vaddress<55>);
```

Library pseudocode for aarch64/exceptions/aborts/BranchTargetException

```
// BranchTargetException()
// =======================
// Raise branch target exception.

AArch64.BranchTargetException(bits(52) vaddress)
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_BranchTarget);
    exception.syndrome<1:0>   = PSTATE.BTYPE;
    exception.syndrome<24:2>  = Zeros(23);         // RES0

    bits(2) target_el = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/aborts/TakeGPCException

```
// TakeGPCException()
// ==================
// Report Granule Protection Exception faults

TakeGPCException(bits(64) vaddress, FaultRecord fault)
    assert HaveRME();
    assert HavePrivATExt();
    assert HaveAtomicExt();
    assert HaveAccessFlagUpdateExt();
    assert HaveDirtyBitModifierExt();
    assert HaveDoubleFaultExt();

    ExceptionRecord exception;

    exception.exceptype = Exception_GPC;
    exception.vaddress  = ZeroExtend(vaddress, 64);
    exception.paddress  = fault.paddress;

    if IPAValid(fault) then
        exception.ipavalid  = TRUE;
        exception.NS        = if fault.ipaddress.paspace == PAS_NonSecure then '1' else '0';
        exception.ipaddress = fault.ipaddress.address;
    else
        exception.ipavalid = FALSE;

    // Populate the fields grouped in ISS
    exception.syndrome<24:22> = Zeros(3); // RES0
    exception.syndrome<21>    = if fault.gpcfs2walk then '1' else '0';  // S2PTW
    exception.syndrome<20>    = if fault.acctype == AccType_IFETCH then '1' else '0'; // InD
    exception.syndrome<19:14> = EncodeGPCSC(fault.gpcf); // GPCSC
    if HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER then
        exception.syndrome<13> = '1'; // VNCR
    else
        exception.syndrome<13> = '0'; // VNCR
    exception.syndrome<12:11> = '00'; // RES0
    exception.syndrome<10:9>  = '00'; // RES0

    if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT, AccType_ATPAN} then
        exception.syndrome<8> = '1';  // CM
    else
        exception.syndrome<8> = '0';  // CM

    exception.syndrome<7> = if fault.s2fs1walk then '1' else '0'; // S1PTW

    if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT, AccType_ATPAN} then
        exception.syndrome<6> = '1';                              // WnR
    elsif fault.statuscode IN {Fault_HWUpdateAccessFlag, Fault_Exclusive} then
        exception.syndrome<6> = bit UNKNOWN;                      // WnR
    elsif IsAtomicRW(fault.acctype) && IsExternalAbort(fault) then
        exception.syndrome<6> = bit UNKNOWN;                      // WnR
    else
        exception.syndrome<6> = if fault.write then '1' else '0'; // WnR

    exception.syndrome<5:0> = EncodeLDFSC(fault.statuscode, fault.level); // xFSC

    bits(64) preferred_exception_return = ThisInstrAddr(64);

    integer vect_offset;
    if IsExternalAbort(fault) && SCR_EL3.EASE == '1' then
        vect_offset = 0x180;
    else
        vect_offset = 0x0;

    AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/async/AArch64.TakePhysicalFIQException

```
// AArch64.TakePhysicalFIQException()
// ==================================

AArch64.TakePhysicalFIQException()

    route_to_el3 = HaveEL(EL3) && SCR_EL3.FIQ == '1';
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || HCR_EL2.FMO == '1'));
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x100;
    exception = ExceptionSyndrome(Exception_FIQ);

    if route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        assert PSTATE.EL IN {EL0, EL1};
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/async/AArch64.TakePhysicalIRQException

```
// AArch64.TakePhysicalIRQException()
// ==================================
// Take an enabled physical IRQ exception.

AArch64.TakePhysicalIRQException()

    route_to_el3 = HaveEL(EL3) && SCR_EL3.IRQ == '1';
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || HCR_EL2.IMO == '1'));
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x80;

    exception = ExceptionSyndrome(Exception_IRQ);

    if route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        assert PSTATE.EL IN {EL0, EL1};
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/async/AArch64.TakePhysicalSErrorException

```
// AArch64.TakePhysicalSErrorException()
// =====================================

AArch64.TakePhysicalSErrorException(bits(25) syndrome)

    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1';
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || (!IsInHost() && HCR_EL2.AMO == '1')));
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x180;

    bits(2) target_el;
    if PSTATE.EL == EL3 || route_to_el3 then
        target_el = EL3;
    elsif PSTATE.EL == EL2 || route_to_el2 then
        target_el = EL2;
    else
        target_el = EL1;

    if IsSErrorEdgeTriggered(target_el, syndrome) then
        ClearPendingPhysicalSError();

    exception = ExceptionSyndrome(Exception_SError);
    exception.syndrome = syndrome;
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/async/AArch64.TakeVirtualFIQException

```
// AArch64.TakeVirtualFIQException()
// =================================

AArch64.TakeVirtualFIQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1';  // Virtual IRQ enabled if TGE==0 and FMO==1

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x100;

    exception = ExceptionSyndrome(Exception_FIQ);

    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/async/AArch64.TakeVirtualIRQException

```
// AArch64.TakeVirtualIRQException()
// =================================

AArch64.TakeVirtualIRQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1';  // Virtual IRQ enabled if TGE==0 and IMO==1

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x80;

    exception = ExceptionSyndrome(Exception_IRQ);

    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/async/AArch64.TakeVirtualSErrorException

```
// AArch64.TakeVirtualSErrorException()
// ====================================

AArch64.TakeVirtualSErrorException(bits(25) syndrome)

    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';  // Virtual SError enabled if TGE==0 and AMO==1

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x180;
    exception = ExceptionSyndrome(Exception_SError);

    if HaveRASExt() then
        exception.syndrome<24>   = VSESR_EL2.IDS;
        exception.syndrome<23:0> = VSESR_EL2.ISS;
    else
        impdef_syndrome = syndrome<24> == '1';
        if impdef_syndrome then exception.syndrome = syndrome;

    ClearPendingVirtualSError();
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/debug/AArch64.BreakpointException

```
// AArch64.BreakpointException()
// =============================

AArch64.BreakpointException(FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    vaddress = bits(64) UNKNOWN;
    exception = AArch64.AbortSyndrome(Exception_Breakpoint, fault, vaddress);

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/debug/AArch64.SoftwareBreakpoint

```
// AArch64.SoftwareBreakpoint()
// ============================

AArch64.SoftwareBreakpoint(bits(16) immediate)

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} &&
                    EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SoftwareBreakpoint);
    exception.syndrome<15:0> = immediate;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/debug/AArch64.SoftwareStepException

```
// AArch64.SoftwareStepException()
// ===============================

AArch64.SoftwareStepException()
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SoftwareStep);
    if SoftwareStep_DidNotStep() then
        exception.syndrome<24> = '0';
    else
        exception.syndrome<24> = '1';
        exception.syndrome<6> = if SoftwareStep_SteppedEX() then '1' else '0';
    exception.syndrome<5:0> = '100010';                // IFSC = Debug Exception

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/debug/AArch64.VectorCatchException

```
// AArch64.VectorCatchException()
// ==============================
// Vector Catch taken from EL0 or EL1 to EL2. This can only be called when debug exceptions are
// being routed to EL2, as Vector Catch is a legacy debug event.

AArch64.VectorCatchException(FaultRecord fault)
    assert PSTATE.EL != EL2;
    assert EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    vaddress = bits(64) UNKNOWN;
    exception = AArch64.AbortSyndrome(Exception_VectorCatch, fault, vaddress);

    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/debug/AArch64.WatchpointException

```
// AArch64.WatchpointException()
// =============================

AArch64.WatchpointException(bits(64) vaddress, FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    ExceptionRecord exception;
    if HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER then
        exception = AArch64.AbortSyndrome(Exception_NV2Watchpoint, fault, vaddress);
    else
        exception = AArch64.AbortSyndrome(Exception_Watchpoint, fault, vaddress);

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/exceptions/AArch64.ExceptionClass

```
// AArch64.ExceptionClass()
// ========================
// Returns the Exception Class and Instruction Length fields to be reported in ESR

(integer,bit) AArch64.ExceptionClass(Exception exceptype, bits(2) target_el)

    il_is_valid = TRUE;
    from_32 = UsingAArch32();
    integer ec;
    case exceptype of
        when Exception_Uncategorized        ec = 0x00; il_is_valid = FALSE;
        when Exception_WFxTrap              ec = 0x01;
        when Exception_CP15RTTrap           ec = 0x03; assert from_32;
        when Exception_CP15RRTTrap          ec = 0x04; assert from_32;
        when Exception_CP14RTTrap           ec = 0x05; assert from_32;
        when Exception_CP14DTTrap           ec = 0x06; assert from_32;
        when Exception_AdvSIMDFPAccessTrap  ec = 0x07;
        when Exception_FPIDTrap             ec = 0x08;
        when Exception_PACTrap              ec = 0x09;
        when Exception_LDST64BTrap          ec = 0x0A;
        when Exception_TSTARTAccessTrap     ec = 0x1B;
        when Exception_GPC                  ec = 0x1E;
        when Exception_CP14RRTTrap          ec = 0x0C; assert from_32;
        when Exception_BranchTarget         ec = 0x0D;
        when Exception_IllegalState         ec = 0x0E; il_is_valid = FALSE;
        when Exception_SupervisorCall       ec = 0x11;
        when Exception_HypervisorCall       ec = 0x12;
        when Exception_MonitorCall          ec = 0x13;
        when Exception_SystemRegisterTrap   ec = 0x18; assert !from_32;
        when Exception_SVEAccessTrap        ec = 0x19; assert !from_32;
        when Exception_ERetTrap             ec = 0x1A; assert !from_32;
        when Exception_PACFail              ec = 0x1C; assert !from_32;
        when Exception_SMEAccessTrap        ec = 0x1D; assert !from_32;
        when Exception_InstructionAbort     ec = 0x20; il_is_valid = FALSE;
        when Exception_PCAlignment          ec = 0x22; il_is_valid = FALSE;
        when Exception_DataAbort            ec = 0x24;
        when Exception_NV2DataAbort         ec = 0x25;
        when Exception_SPAlignment          ec = 0x26; il_is_valid = FALSE; assert !from_32;
        when Exception_MemCpyMemSet         ec = 0x27;
        when Exception_FPTrappedException   ec = 0x28;
        when Exception_SError               ec = 0x2F; il_is_valid = FALSE;
        when Exception_Breakpoint           ec = 0x30; il_is_valid = FALSE;
        when Exception_SoftwareStep         ec = 0x32; il_is_valid = FALSE;
        when Exception_Watchpoint           ec = 0x34; il_is_valid = FALSE;
        when Exception_NV2Watchpoint        ec = 0x35; il_is_valid = FALSE;
        when Exception_SoftwareBreakpoint   ec = 0x38;
        when Exception_VectorCatch          ec = 0x3A; il_is_valid = FALSE; assert from_32;
        otherwise                           Unreachable();

    if ec IN {0x20,0x24,0x30,0x32,0x34} && target_el == PSTATE.EL then
        ec = ec + 1;

    if ec IN {0x11,0x12,0x13,0x28,0x38} && !from_32 then
        ec = ec + 4;
    bit il;
    if il_is_valid then
        il = if ThisInstrLength() == 32 then '1' else '0';
    else
        il = '1';
    assert from_32 || il == '1';            // AArch64 instructions always 32-bit

    return (ec,il);
```

Library pseudocode for aarch64/exceptions/exceptions/AArch64.ReportException

```
// AArch64.ReportException()
// =========================
// Report syndrome information for exception taken to AArch64 state.

AArch64.ReportException(ExceptionRecord exception, bits(2) target_el)

    Exception exceptype = exception.exceptype;

    (ec,il) = AArch64.ExceptionClass(exceptype, target_el);
    iss = exception.syndrome;
    iss2 = exception.syndrome2;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';

    ESR[target_el] = (Zeros(27) :   // <63:37>
                      iss2      :   // <36:32>
                      ec<5:0>   :   // <31:26>
                      il        :   // <25>
                      iss);         // <24:0>

    if exceptype IN {
        Exception_InstructionAbort,
        Exception_PCAlignment,
        Exception_DataAbort,
        Exception_NV2DataAbort,
        Exception_NV2Watchpoint,
        Exception_GPC,
        Exception_Watchpoint
    } then
        FAR[target_el] = exception.vaddress;
    else
        FAR[target_el] = bits(64) UNKNOWN;

    if exception.ipavalid then
        HPFAR_EL2<43:4> = exception.ipaddress<51:12>;
        if IsSecureEL2Enabled() && CurrentSecurityState() == SS_Secure then
            HPFAR_EL2.NS = exception.NS;
        else
            HPFAR_EL2.NS = '0';
    elsif target_el == EL2 then
        HPFAR_EL2<43:4> = bits(40) UNKNOWN;

    if exception.exceptype == Exception_GPC then
        MFAR_EL3.FPA = ZeroExtend(exception.paddress.address<AArch64.PAMax()-1:12>, 40);
        case exception.paddress.paspace of
            when PAS_Secure     MFAR_EL3.<NSE,NS> = '00';
            when PAS_NonSecure  MFAR_EL3.<NSE,NS> = '01';
            when PAS_Root       MFAR_EL3.<NSE,NS> = '10';
            when PAS_Realm      MFAR_EL3.<NSE,NS> = '11';
    return;
```

Library pseudocode for aarch64/exceptions/exceptions/AArch64.ResetControlRegisters

```
// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.
AArch64.ResetControlRegisters(boolean cold_reset);
```

Library pseudocode for aarch64/exceptions/exceptions/AArch64.TakeReset

```
// AArch64.TakeReset()
// ===================
// Reset into AArch64 state

AArch64.TakeReset(boolean cold_reset)
    assert HaveAArch64();

    // Enter the highest implemented Exception level in AArch64 state
    PSTATE.nRW = '0';
    if HaveEL(EL3) then
        PSTATE.EL = EL3;
    elsif HaveEL(EL2) then
        PSTATE.EL = EL2;
    else
        PSTATE.EL = EL1;

    // Reset System registers and other system components
    AArch64.ResetControlRegisters(cold_reset);

    // Reset all other PSTATE fields
    PSTATE.SP = '1';              // Select stack pointer
    PSTATE.<D,A,I,F>  = '1111';   // All asynchronous exceptions masked
    PSTATE.SS = '0';              // Clear software step bit
    PSTATE.DIT = '0';             // PSTATE.DIT is reset to 0 when resetting into AArch64
    PSTATE.IL = '0';              // Clear Illegal Execution state bit

    TSTATE.depth = 0;             // Non-transactional state

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // ELR_ELx and SPSR_ELx have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch64.ResetGeneralRegisters();
    AArch64.ResetSIMDFPRegisters();
    AArch64.ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);

    bits(64) rv;                      // IMPLEMENTATION DEFINED reset vector

    if HaveEL(EL3) then
        rv = RVBAR_EL3;
    elsif HaveEL(EL2) then
        rv = RVBAR_EL2;
    else
        rv = RVBAR_EL1;

    // The reset vector must be correctly aligned
    assert IsZero(rv<63:AArch64.PAMax()>) && IsZero(rv<1:0>);

    boolean branch_conditional = FALSE;
    BranchTo(rv, BranchType_RESET, branch_conditional);
```

Library pseudocode for aarch64/exceptions/ieeefp/AArch64.FPTrappedException

```
// AArch64.FPTrappedException()
// ============================

AArch64.FPTrappedException(boolean is_ase, bits(8) accumulated_exceptions)
    exception = ExceptionSyndrome(Exception_FPTrappedException);
    if is_ase then
        if boolean IMPLEMENTATION_DEFINED "vector instructions set TFV to 1" then
            exception.syndrome<23> = '1';                           // TFV
        else
            exception.syndrome<23> = '0';                           // TFV
    else
        exception.syndrome<23> = '1';                               // TFV
    exception.syndrome<10:8> = bits(3) UNKNOWN;                     // VECITR
    if exception.syndrome<23> == '1' then
        exception.syndrome<7,4:0> = accumulated_exceptions<7,4:0>; // IDF,IXF,UFF,OFF,DZF,IOF
    else
        exception.syndrome<7,4:0> = bits(6) UNKNOWN;

    route_to_el2 = EL2Enabled() && HCR_EL2.TGE == '1';

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallHypervisor

```
// AArch64.CallHypervisor()
// ========================
// Performs a HVC call

AArch64.CallHypervisor(bits(16) immediate)
    assert HaveEL(EL2);

    if UsingAArch32() then AArch32.ITAdvance();
    SSAdvance();
    bits(64) preferred_exception_return = NextInstrAddr(64);
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_HypervisorCall);
    exception.syndrome<15:0> = immediate;

    if PSTATE.EL == EL3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallSecureMonitor

```
// AArch64.CallSecureMonitor()
// ===========================

AArch64.CallSecureMonitor(bits(16) immediate)
    assert HaveEL(EL3) && !ELUsingAArch32(EL3);
    if UsingAArch32() then AArch32.ITAdvance();
    SSAdvance();
    bits(64) preferred_exception_return = NextInstrAddr(64);
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_MonitorCall);
    exception.syndrome<15:0> = immediate;

    AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallSupervisor

```
// AArch64.CallSupervisor()
// ========================
// Calls the Supervisor

AArch64.CallSupervisor(bits(16) immediate_in)
    bits(16) immediate = immediate_in;
    if UsingAArch32() then AArch32.ITAdvance();
    SSAdvance();
    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';

    bits(64) preferred_exception_return = NextInstrAddr(64);
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SupervisorCall);
    exception.syndrome<15:0> = immediate;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/takeexception/AArch64.TakeException

```
// AArch64.TakeException()
// =======================
// Take an exception to an Exception level using AArch64.

AArch64.TakeException(bits(2) target_el, ExceptionRecord exception_in,
                      bits(64) preferred_exception_return, integer vect_offset_in)
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);
    ExceptionRecord exception = exception_in;
    boolean sync_errors;
    boolean iesb_req;
    if HaveIESB() then
        sync_errors = SCTLR[target_el].IESB == '1';
        if HaveDoubleFaultExt() then
            sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' && target_el == EL3);
        if sync_errors && InsertIESBBeforeException(target_el) then
            SynchronizeErrors();
            iesb_req = FALSE;
            sync_errors = FALSE;
            TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
    else
        sync_errors = FALSE;

    if HaveTME() && TSTATE.depth > 0 then
        TMFailure cause;
        case exception.exceptype of
            when Exception_SoftwareBreakpoint cause = TMFailure_DBG;
            when Exception_Breakpoint         cause = TMFailure_DBG;
            when Exception_Watchpoint         cause = TMFailure_DBG;
            when Exception_SoftwareStep       cause = TMFailure_DBG;
            otherwise                         cause = TMFailure_ERR;
        FailTransaction(cause, FALSE);

    SynchronizeContext();

    // If coming from AArch32 state, the top parts of the X[] registers might be set to zero
    from_32 = UsingAArch32();
    if from_32 then AArch64.MaybeZeroRegisterUppers();
    if from_32 && HaveSME() && PSTATE.SM == '1' then
        ResetSVEState();
    else
        MaybeZeroSVEUppers(target_el);

    integer vect_offset = vect_offset_in;
    if UInt(target_el) > UInt(PSTATE.EL) then
        boolean lower_32;
        if target_el == EL3 then
            if EL2Enabled() then
                lower_32 = ELUsingAArch32(EL2);
            else
                lower_32 = ELUsingAArch32(EL1);
        elsif IsInHost() && PSTATE.EL == EL0 && target_el == EL2 then
            lower_32 = ELUsingAArch32(EL0);
        else
            lower_32 = ELUsingAArch32(target_el - 1);
        vect_offset = vect_offset + (if lower_32 then 0x600 else 0x400);

    elsif PSTATE.SP == '1' then
        vect_offset = vect_offset + 0x200;

    bits(64) spsr = GetPSRFromPSTATE(AArch64_NonDebugState, 64);

    if PSTATE.EL == EL1 && target_el == EL1 && EL2Enabled() then
        if HaveNV2Ext() && (HCR_EL2.<NV,NV1,NV2> == '100' || HCR_EL2.<NV,NV1,NV2> == '111') then
            spsr<3:2> = '10';
        else
            if HaveNVExt() && HCR_EL2.<NV,NV1> == '10' then
                spsr<3:2> = '10';

    if HaveBTIExt() && !UsingAArch32() then
        boolean zero_btype;
```

```
    // SPSR[].BTYPE is only guaranteed valid for these exception types
    if exception.exceptype IN {Exception_SError, Exception_IRQ, Exception_FIQ,
                           Exception_SoftwareStep, Exception_PCAlignment,
                           Exception_InstructionAbort, Exception_Breakpoint,
                           Exception_VectorCatch, Exception_SoftwareBreakpoint,
                           Exception_IllegalState, Exception_BranchTarget} then
        zero_btype = FALSE;
    else
        zero_btype = ConstrainUnpredictableBool(Unpredictable_ZEROBTYPE);
    if zero_btype then spsr<11:10> = '00';

if HaveNV2Ext() && exception.exceptype == Exception_NV2DataAbort && target_el == EL3 then
    // External aborts are configured to be taken to EL3
    exception.exceptype = Exception_DataAbort;
if !(exception.exceptype IN {Exception_IRQ, Exception_FIQ}) then
    AArch64.ReportException(exception, target_el);

if HaveBRBExt() then
    BRBEException(exception.exceptype, preferred_exception_return,
                  VBAR[target_el]<63:11>:vect_offset<10:0>, target_el,
                  exception.trappedsyscallinst);

PSTATE.EL = target_el;
PSTATE.nRW = '0';
PSTATE.SP = '1';

SPSR[] = spsr;
ELR[] = preferred_exception_return;

PSTATE.SS = '0';
if HaveFeatNMI() && !ELUsingAArch32(target_el) then PSTATE.ALLINT = NOT SCTLR[].SPINTMASK;
PSTATE.<D,A,I,F> = '1111';
PSTATE.IL = '0';
if from_32 then                             // Coming from AArch32
    PSTATE.IT = '00000000';
    PSTATE.T = '0';                         // PSTATE.J is RES0
if (HavePANExt() && (PSTATE.EL == EL1 || (PSTATE.EL == EL2 && ELIsInHost(EL0))) &&
    SCTLR[].SPAN == '0') then
    PSTATE.PAN = '1';
if HaveUAOExt() then PSTATE.UAO = '0';
if HaveBTIExt() then PSTATE.BTYPE = '00';
if HaveSSBSExt() then PSTATE.SSBS = SCTLR[].DSSBS;
if HaveMTEExt() then PSTATE.TCO = '1';

boolean branch_conditional = FALSE;
BranchTo(VBAR[]<63:11>:vect_offset<10:0>, BranchType_EXCEPTION, branch_conditional);

CheckExceptionCatch(TRUE);                  // Check for debug event on exception entry

if sync_errors then
    SynchronizeErrors();
    iesb_req = TRUE;
    TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);

EndOfInstruction();
```

Library pseudocode for aarch64/exceptions/traps/AArch64.AArch32SystemAccessTrap

```
// AArch64.AArch32SystemAccessTrap()
// =================================
// Trapped AARCH32 System register access.

AArch64.AArch32SystemAccessTrap(bits(2) target_el, integer ec)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    exception = AArch64.AArch32SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.AArch32SystemAccessTrapSyndrome

```
// AArch64.AArch32SystemAccessTrapSyndrome()
// =========================================
// Returns the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS, VMSR instructior
// other than traps that are due to HCPTR or CPACR.

ExceptionRecord AArch64.AArch32SystemAccessTrapSyndrome(bits(32) instr, integer ec)
    ExceptionRecord exception;

    case ec of
        when 0x0    exception = ExceptionSyndrome(Exception_Uncategorized);
        when 0x3    exception = ExceptionSyndrome(Exception_CP15RTTrap);
        when 0x4    exception = ExceptionSyndrome(Exception_CP15RRTTrap);
        when 0x5    exception = ExceptionSyndrome(Exception_CP14RTTrap);
        when 0x6    exception = ExceptionSyndrome(Exception_CP14DTTrap);
        when 0x7    exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        when 0x8    exception = ExceptionSyndrome(Exception_FPIDTrap);
        when 0xC    exception = ExceptionSyndrome(Exception_CP14RRTTrap);
        otherwise   Unreachable();

    bits(20) iss = Zeros(20);

    if exception.exceptype == Exception_Uncategorized then
        return exception;
    elsif exception.exceptype IN {Exception_FPIDTrap, Exception_CP14RTTrap, Exception_CP15RTTrap} then
        // Trapped MRC/MCR, VMRS on FPSID
        if exception.exceptype != Exception_FPIDTrap then    // When trap is not for VMRS
            iss<19:17> = instr<7:5>;           // opc2
            iss<16:14> = instr<23:21>;         // opc1
            iss<13:10> = instr<19:16>;         // CRn
            iss<4:1>   = instr<3:0>;           // CRm
        else
            iss<19:17> = '000';
            iss<16:14> = '111';
            iss<13:10> = instr<19:16>;         // reg
            iss<4:1>   = '0000';

        if instr<20> == '1' && instr<15:12> == '1111' then    // MRC, Rt==15
            iss<9:5> = '11111';
        elsif instr<20> == '0' && instr<15:12> == '1111' then // MCR, Rt==15
            iss<9:5> = bits(5) UNKNOWN;
        else
            iss<9:5> = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
    elsif exception.exceptype IN {Exception_CP14RRTTrap, Exception_AdvSIMDFPAccessTrap, Exception_CP15RRT
        // Trapped MRRC/MCRR, VMRS/VMSR
        iss<19:16> = instr<7:4>;          // opc1
        if instr<19:16> == '1111' then    // Rt2==15
            iss<14:10> = bits(5) UNKNOWN;
        else
            iss<14:10> = LookUpRIndex(UInt(instr<19:16>), PSTATE.M)<4:0>;

        if instr<15:12> == '1111' then    // Rt==15
            iss<9:5> = bits(5) UNKNOWN;
        else
            iss<9:5> = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
        iss<4:1>   = instr<3:0>;          // CRm
    elsif exception.exceptype == Exception_CP14DTTrap then
        // Trapped LDC/STC
        iss<19:12> = instr<7:0>;          // imm8
        iss<4>     = instr<23>;           // U
        iss<2:1>   = instr<24,21>;        // P,W
        if instr<19:16> == '1111' then    // Rn==15, LDC(Literal addressing)/STC
            iss<9:5> = bits(5) UNKNOWN;
            iss<3>   = '1';
    iss<0> = instr<20>;                   // Direction

    exception.syndrome<24:20> = ConditionSyndrome();
    exception.syndrome<19:0>  = iss;

    return exception;
```

Library pseudocode for aarch64/exceptions/traps/AArch64.AdvSIMDFPAccessTrap

```
// AArch64.AdvSIMDFPAccessTrap()
// ============================
// Trapped access to Advanced SIMD or FP registers due to CPACR[].

AArch64.AdvSIMDFPAccessTrap(bits(2) target_el)
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    ExceptionRecord exception;
    vect_offset = 0x0;

    route_to_el2 = (target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1');

    if route_to_el2 then
        exception = ExceptionSyndrome(Exception_Uncategorized);
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        exception.syndrome<24:20> = ConditionSyndrome();
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

    return;
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckCP15InstrCoarseTraps

```
// AArch64.CheckCP15InstrCoarseTraps()
// ===================================
// Check for coarse-grained AArch32  traps to System registers in the
// coproc=0b1111 encoding space by HSTR_EL2, HCR_EL2, and SCTLR_ELx.

AArch64.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)
    trapped_encoding = ((CRn == 9  && CRm IN {0,1,2,    5,6,7,8   }) ||
                        (CRn == 10 && CRm IN {0,1,    4,      8   }) ||
                        (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}));

    // Check for MRC and MCR disabled by SCTLR_EL1.TIDCP.
    if (HaveFeatTIDCP1() && PSTATE.EL == EL0 && !IsInHost() &&
           !ELUsingAArch32(EL1) && SCTLR_EL1.TIDCP == '1' && trapped_encoding) then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x3);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x3);

    // Check for coarse-grained Hyp traps
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        // Check for MRC and MCR disabled by SCTLR_EL2.TIDCP.
        if (HaveFeatTIDCP1() && PSTATE.EL == EL0 && IsInHost() &&
                SCTLR_EL2.TIDCP == '1' && trapped_encoding) then
            AArch64.AArch32SystemAccessTrap(EL2, 0x3);

        major = if nreg == 1 then CRn else CRm;
        // Check for MCR, MRC, MCRR, and MRRC disabled by HSTR_EL2<CRn/CRm>
        // and MRC and MCR disabled by HCR_EL2.TIDCP.
        if ((!IsInHost() && !(major IN {4,14}) && HSTR_EL2<major> == '1') ||
                (HCR_EL2.TIDCP == '1' && nreg == 1 && trapped_encoding)) then
            if (PSTATE.EL == EL0 &&
                    boolean IMPLEMENTATION_DEFINED "UNDEF unallocated CP15 access at EL0") then
                UNDEFINED;
            AArch64.AArch32SystemAccessTrap(EL2, 0x3);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDEnabled

```
// AArch64.CheckFPAdvSIMDEnabled()
// ===============================

AArch64.CheckFPAdvSIMDEnabled()
    AArch64.CheckFPEnabled();
    // Check for illegal use of Advanced
    // SIMD in Streaming SVE Mode
    if HaveSME() && PSTATE.SM == '1' && !IsFullA64Enabled() then
        SMEAccessTrap(SMEExceptionType_Streaming, PSTATE.EL);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDTrap

```
// AArch64.CheckFPAdvSIMDTrap()
// ============================
// Check against CPTR_EL2 and CPTR_EL3.

AArch64.CheckFPAdvSIMDTrap()
    if HaveEL(EL3) && CPTR_EL3.TFP == '1' && EL3SDDTrapPriority() then
        UNDEFINED;

    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        // Check if access disabled in CPTR_EL2
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            boolean disabled;
            case CPTR_EL2.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
        else
            if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);

    if HaveEL(EL3) then
        // Check if access disabled in CPTR_EL3
        if CPTR_EL3.TFP == '1' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.AdvSIMDFPAccessTrap(EL3);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckFPEnabled

```
// AArch64.CheckFPEnabled()
// ========================
// Check against CPACR[]

AArch64.CheckFPEnabled()
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check if access disabled in CPACR_EL1
        boolean disabled;
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

    AArch64.CheckFPAdvSIMDTrap();               // Also check against CPTR_EL2 and CPTR_EL3
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForERetTrap

```
// AArch64.CheckForERetTrap()
// ==========================
// Check for trap on ERET, ERETAA, ERETAB instruction

AArch64.CheckForERetTrap(boolean eret_with_pac, boolean pac_uses_key_a)

    route_to_el2 = FALSE;
    // Non-secure EL1 execution of ERET, ERETAA, ERETAB when either HCR_EL2.NV or HFGITR_EL2.ERET is set,
    // is trapped to EL2
    route_to_el2 = (PSTATE.EL == EL1 && EL2Enabled() &&
                    ((HaveNVExt() && HCR_EL2.NV == '1') ||
                    (HaveFGTExt() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
                    HFGITR_EL2.ERET == '1')));
    if route_to_el2 then
        ExceptionRecord exception;
        bits(64) preferred_exception_return = ThisInstrAddr(64);
        vect_offset = 0x0;
        exception = ExceptionSyndrome(Exception_ERetTrap);
        if !eret_with_pac then                              // ERET
            exception.syndrome<1> = '0';
            exception.syndrome<0> = '0';                    // RES0
        else
            exception.syndrome<1> = '1';
            if pac_uses_key_a then                          // ERETAA
                exception.syndrome<0> = '0';
            else    // ERETAB
                exception.syndrome<0> = '1';
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForSMCUndefOrTrap

```
// AArch64.CheckForSMCUndefOrTrap()
// ================================
// Check for UNDEFINED or trap on SMC instruction

AArch64.CheckForSMCUndefOrTrap(bits(16) imm)
    if PSTATE.EL == EL0 then UNDEFINED;
    if (!(PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1') &&
        HaveEL(EL3) && SCR_EL3.SMD == '1') then
        UNDEFINED;
    route_to_el2 = FALSE;
    if !HaveEL(EL3) then
        if PSTATE.EL == EL1 && EL2Enabled() then
            if HaveNVExt() && HCR_EL2.NV == '1' && HCR_EL2.TSC == '1' then
                route_to_el2 = TRUE;
            else
                UNDEFINED;
        else
            UNDEFINED;
    else
        route_to_el2 = PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1';
    if route_to_el2 then
        bits(64) preferred_exception_return = ThisInstrAddr(64);
        vect_offset = 0x0;
        exception = ExceptionSyndrome(Exception_MonitorCall);
        exception.syndrome<15:0> = imm;
        exception.trappedsyscallinst = TRUE;
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForSVCTrap

```
// AArch64.CheckForSVCTrap()
// =========================
// Check for trap on SVC instruction

AArch64.CheckForSVCTrap(bits(16) immediate)
    if HaveFGTExt() then
        route_to_el2 = FALSE;
        if PSTATE.EL == EL0 then
            route_to_el2 = (!UsingAArch32() && !ELUsingAArch32(EL1) &&
                            EL2Enabled() && HFGITR_EL2.SVC_EL0 == '1' &&
                            (HCR_EL2.<E2H, TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')));

        elsif PSTATE.EL == EL1 then
            route_to_el2 = (EL2Enabled() && HFGITR_EL2.SVC_EL1 == '1' &&
                            (!HaveEL(EL3) || SCR_EL3.FGTEn == '1'));

        if route_to_el2 then
            exception = ExceptionSyndrome(Exception_SupervisorCall);
            exception.syndrome<15:0> = immediate;
            exception.trappedsyscallinst = TRUE;
            bits(64) preferred_exception_return = ThisInstrAddr(64);
            vect_offset = 0x0;

            AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForWFxTrap

```
// AArch64.CheckForWFxTrap()
// =========================
// Check for trap on WFE or WFI instruction

AArch64.CheckForWFxTrap(bits(2) target_el, WFxType wfxtype)
    assert HaveEL(target_el);

    boolean is_wfe = wfxtype IN {WFxType_WFE, WFxType_WFET};
    boolean trap;
    case target_el of
        when EL1
            trap = (if is_wfe then SCTLR[].nTWE else SCTLR[].nTWI) == '0';
        when EL2
            trap = (if is_wfe then HCR_EL2.TWE else HCR_EL2.TWI) == '1';
        when EL3
            trap = (if is_wfe then SCR_EL3.TWE else SCR_EL3.TWI) == '1';

    if trap then
        AArch64.WFxTrap(wfxtype, target_el);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckIllegalState

```
// AArch64.CheckIllegalState()
// ===========================
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

AArch64.CheckIllegalState()
    if PSTATE.IL == '1' then
        route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';

        bits(64) preferred_exception_return = ThisInstrAddr(64);
        vect_offset = 0x0;

        exception = ExceptionSyndrome(Exception_IllegalState);

        if UInt(PSTATE.EL) > UInt(EL1) then
            AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
        elsif route_to_el2 then
            AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
        else
            AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.MonitorModeTrap

```
// AArch64.MonitorModeTrap()
// =========================
// Trapped use of Monitor mode features in a Secure EL1 AArch32 mode

AArch64.MonitorModeTrap()
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_Uncategorized);

    if IsSecureEL2Enabled() then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.SystemAccessTrap

```
// AArch64.SystemAccessTrap()
// ==========================
// Trapped access to AArch64 System register or system instruction.

AArch64.SystemAccessTrap(bits(2) target_el, integer ec)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    exception = AArch64.SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.SystemAccessTrapSyndrome

```
// AArch64.SystemAccessTrapSyndrome()
// ==================================
// Returns the syndrome information for traps on AArch64 MSR/MRS instructions.

ExceptionRecord AArch64.SystemAccessTrapSyndrome(bits(32) instr_in, integer ec)
    ExceptionRecord exception;
    bits(32) instr = instr_in;
    case ec of
        when 0x0                                                    // Trapped access due to unknown rea
            exception = ExceptionSyndrome(Exception_Uncategorized);
        when 0x7                                                    // Trapped access to SVE, Advance SI
            exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
            exception.syndrome<24:20> = ConditionSyndrome();
        when 0x18                                                   // Trapped access to System registe
            exception = ExceptionSyndrome(Exception_SystemRegisterTrap);
            instr = ThisInstr();
            exception.syndrome<21:20> = instr<20:19>;          // Op0
            exception.syndrome<19:17> = instr<7:5>;            // Op2
            exception.syndrome<16:14> = instr<18:16>;          // Op1
            exception.syndrome<13:10> = instr<15:12>;          // CRn
            exception.syndrome<9:5>   = instr<4:0>;            // Rt
            exception.syndrome<4:1>   = instr<11:8>;           // CRm
            exception.syndrome<0>     = instr<21>;             // Direction
        when 0x19                                                   // Trapped access to SVE System regi
            exception = ExceptionSyndrome(Exception_SVEAccessTrap);
        when 0x1D                                                   // Trapped access to SME System regi
            exception = ExceptionSyndrome(Exception_SMEAccessTrap);
        otherwise
            Unreachable();

    return exception;
```

Library pseudocode for aarch64/exceptions/traps/AArch64.UndefinedFault

```
// AArch64.UndefinedFault()
// ========================

AArch64.UndefinedFault()

    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_Uncategorized);

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.WFxTrap

```
// AArch64.WFxTrap()
// =================

AArch64.WFxTrap(WFxType wfxtype, bits(2) target_el)
    assert UInt(target_el) > UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_WFxTrap);
    exception.syndrome<24:20> = ConditionSyndrome();

    case wfxtype of
        when WFxType_WFI
            exception.syndrome<1:0> = '00';
        when WFxType_WFE
            exception.syndrome<1:0> = '01';
        when WFxType_WFIT
            exception.syndrome<1:0> = '10';
            exception.syndrome<2>   = '1';   // Register field is valid
            exception.syndrome<9:5> = ThisInstr()<4:0>;
        when WFxType_WFET
            exception.syndrome<1:0> = '11';
            exception.syndrome<2>   = '1';   // Register field is valid
            exception.syndrome<9:5> = ThisInstr()<4:0>;

    if target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/CheckFPAdvSIMDEnabled64

```
// CheckFPAdvSIMDEnabled64()
// =========================
// AArch64 instruction wrapper

CheckFPAdvSIMDEnabled64()
    AArch64.CheckFPAdvSIMDEnabled();
```

Library pseudocode for aarch64/exceptions/traps/CheckFPEnabled64

```
// CheckFPEnabled64()
// ==================
// AArch64 instruction wrapper

CheckFPEnabled64()
    AArch64.CheckFPEnabled();
```

Library pseudocode for aarch64/exceptions/traps/CheckLDST64BEnabled

```
// CheckLDST64BEnabled()
// =====================
// Checks for trap on ST64B and LD64B instructions

CheckLDST64BEnabled()
    boolean trap = FALSE;
    bits(25) iss = ZeroExtend('10', 25);  // 0x2
    bits(2) target_el;

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTLR_EL1.EnALS == '0';
            target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
        else
            trap = SCTLR_EL2.EnALS == '0';
            target_el = EL2;
    else
        target_el = EL1;

    if (!trap && EL2Enabled() && HaveFeatHCX() &&
        ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
        trap = !IsHCRXEL2Enabled() || HCRX_EL2.EnALS == '0';
        target_el = EL2;

    if trap then LDST64BTrap(target_el, iss);
```

Library pseudocode for aarch64/exceptions/traps/CheckST64BV0Enabled

```
// CheckST64BV0Enabled()
// =====================
// Checks for trap on ST64BV0 instruction

CheckST64BV0Enabled()
    boolean trap = FALSE;
    bits(25) iss = ZeroExtend('1', 25);  // 0x1
    bits(2) target_el;

    if (PSTATE.EL != EL3 && HaveEL(EL3) &&
        SCR_EL3.EnAS0 == '0' && EL3SDDTrapPriority()) then
        UNDEFINED;

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTLR_EL1.EnAS0 == '0';
            target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
        else
            trap = SCTLR_EL2.EnAS0 == '0';
            target_el = EL2;

    if (!trap && EL2Enabled() && HaveFeatHCX() &&
        ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
        trap = !IsHCRXEL2Enabled() || HCRX_EL2.EnAS0 == '0';
        target_el = EL2;

    if !trap && PSTATE.EL != EL3 then
        trap = HaveEL(EL3) && SCR_EL3.EnAS0 == '0';
        target_el = EL3;

    if trap then
        if target_el == EL3 && Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            LDST64BTrap(target_el, iss);
```

Library pseudocode for aarch64/exceptions/traps/CheckST64BVEnabled

```
// CheckST64BVEnabled()
// ====================
// Checks for trap on ST64BV instruction

CheckST64BVEnabled()
    boolean trap = FALSE;
    bits(25) iss = Zeros(25);
    bits(2) target_el;

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTLR_EL1.EnASR == '0';
            target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
        else
            trap = SCTLR_EL2.EnASR == '0';
            target_el = EL2;

    if (!trap && EL2Enabled() && HaveFeatHCX() &&
        ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
        trap = !IsHCRXEL2Enabled() || HCRX_EL2.EnASR == '0';
        target_el = EL2;

    if trap then LDST64BTrap(target_el, iss);
```

Library pseudocode for aarch64/exceptions/traps/LDST64BTrap

```
// LDST64BTrap()
// =============
// Trapped access to LD64B, ST64B, ST64BV and ST64BV0 instructions

LDST64BTrap(bits(2) target_el, bits(25) iss)
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_LDST64BTrap);
    exception.syndrome = iss;
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

    return;
```

Library pseudocode for aarch64/exceptions/traps/WFETrapDelay

```
// WFETrapDelay()
// ==============
// Returns TRUE when delay in trap to WFE is enabled with value to amount of delay,
// FALSE otherwise.

(boolean, integer) WFETrapDelay(bits(2) target_el)
    boolean delay_enabled;
    integer delay;
    case target_el of
        when EL1
            if !IsInHost() then
                delay_enabled = SCTLR_EL1.TWEDEn == '1';
                delay         = 1 << (UInt(SCTLR_EL1.TWEDEL) + 8);
            else
                delay_enabled = SCTLR_EL2.TWEDEn == '1';
                delay         = 1 << (UInt(SCTLR_EL2.TWEDEL) + 8);
        when EL2
            assert EL2Enabled();
            delay_enabled = HCR_EL2.TWEDEn == '1';
            delay         = 1 << (UInt(HCR_EL2.TWEDEL) + 8);
        when EL3
            delay_enabled = SCR_EL3.TWEDEn == '1';
            delay         = 1 << (UInt(SCR_EL3.TWEDEL) + 8);

    return (delay_enabled, delay);
```

Library pseudocode for aarch64/exceptions/traps/WaitForEventUntilDelay

```
// Returns TRUE if WaitForEvent() returns before WFE trap delay expires,
// FALSE otherwise.
boolean WaitForEventUntilDelay(boolean delay_enabled, integer delay);
```

Library pseudocode for aarch64/functions/aborts/AArch64.FaultSyndrome

```
// AArch64.FaultSyndrome()
// =======================
// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to
// an Exception level using AArch64.

(bits(25), bits(5)) AArch64.FaultSyndrome(boolean d_side, FaultRecord fault)
    assert fault.statuscode != Fault_None;

    bits(25) iss = Zeros(25);
    bits(5) iss2 = Zeros(5);

    if HaveRASExt() && fault.statuscode == Fault_SyncExternal then
        iss<12:11> = fault.errortype; // SET
    if d_side then
        if HaveFeatLS64() && fault.acctype == AccType_ATOMICLS64 then
            if (fault.statuscode IN {Fault_AccessFlag, Fault_Translation,
                                     Fault_Permission}) then
                (iss2, iss<24:14>) = LS64InstructionSyndrome();
        elsif (IsSecondStage(fault) && !fault.s2fs1walk &&
               (!IsExternalSyncAbort(fault) ||
               (!HaveRASExt() && fault.acctype == AccType_TTW &&
               boolean IMPLEMENTATION_DEFINED "ISV on second stage translation table walk"))) then
            iss<24:14> = LSInstructionSyndrome();

        if HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER then
            iss<13> = '1';   // Fault is generated by use of VNCR_EL2

        if HaveFeatLS64() && fault.statuscode IN {Fault_AccessFlag, Fault_Translation,
                                                  Fault_Permission} then
            iss<12:11> = GetLoadStoreType();
        if fault.acctype IN {AccType_DC, AccType_IC, AccType_AT, AccType_ATPAN} then
            iss<8> = '1';  iss<6> = '1';
        else
            iss<6> = if fault.write then '1' else '0';

    if IsExternalAbort(fault) then iss<9> = fault.extflag;
    iss<7> = if fault.s2fs1walk then '1' else '0';
    iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);

    return (iss, iss2);

bits(6) EncodeGPCSC(GPCFRecord gpcf)
    assert gpcf.level IN {0,1};

    case gpcf.gpf of
        when GPCF_AddressSize return '00000':gpcf.level<0>;
        when GPCF_Walk        return '00010':gpcf.level<0>;
        when GPCF_Fail        return '00110':gpcf.level<0>;
        when GPCF_EABT        return '01010':gpcf.level<0>;
```

Library pseudocode for aarch64/functions/aborts/LS64InstructionSyndrome

```
// Returns the syndrome information and LST for a Data Abort by a
// ST64B, ST64BV, ST64BV0, or LD64B instruction. The syndrome information
// includes the ISS2, extended syndrome field.
(bits(5), bits(11)) LS64InstructionSyndrome();
```

Library pseudocode for aarch64/functions/cache/AArch64.DataMemZero

```
// AArch64.DataMemZero()
// =====================
// Write Zero to data memory.

AArch64.DataMemZero(bits(64) regval, bits(64) vaddress, AddressDescriptor memaddrdesc_in, integer size)
    iswrite = TRUE;
    AddressDescriptor memaddrdesc = memaddrdesc_in;
    accdesc = CreateAccessDescriptor(AccType_DCZVA);
    if HaveTME() then
        accdesc.transactional = TSTATE.depth > 0;
        if accdesc.transactional && !MemHasTransactionalAccess(memaddrdesc.memattrs) then
            FailTransaction(TMFailure_IMP, FALSE);

    for i = 0 to size-1
        if HaveMTE2Ext() then
            if AArch64.AccessIsTagChecked(vaddress, AccType_DCZVA) then
                bits(4) ptag = AArch64.PhysicalTag(vaddress);
                if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
                    if boolean IMPLEMENTATION_DEFINED "DC_ZVA tag fault reported with lowest faulting add
                        AArch64.TagCheckFault(vaddress, AccType_DCZVA, iswrite);
                    else
                        AArch64.TagCheckFault(regval, AccType_DCZVA, iswrite);

        memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, Zeros(8));
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
        memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
    return;
```

Library pseudocode for aarch64/functions/cache/AArch64.TagMemZero

```
// AArch64.TagMemZero()
// ====================
// Write Zero to tag memory.

AArch64.TagMemZero(bits(64) vaddress_in, integer size)
    bits(64) vaddress = vaddress_in;
    integer count = size >> LOG2_TAG_GRANULE;
    bits(4) tag = AArch64.AllocationTagFromAddress(vaddress);
    for i = 0 to count-1
        AArch64.MemTag[vaddress, AccType_NORMAL] = tag;
        vaddress = vaddress + TAG_GRANULE;
    return;
```

Library pseudocode for aarch64/functions/exclusive/AArch64.ExclusiveMonitorsPass

```
// AArch64.ExclusiveMonitorsPass()
// ===============================
// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

boolean AArch64.ExclusiveMonitorsPass(bits(64) address, integer size)

    // It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
    // before or after the check on the local Exclusives monitor. As a result a failure
    // of the local monitor can occur on some implementations even if the memory
    // access would give an memory abort.

    acctype = AccType_ATOMIC;
    iswrite = TRUE;

    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);

    passed = AArch64.IsExclusiveVA(address, ProcessorID(), size);
    if !passed then
        return FALSE;

    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
    ClearExclusiveLocal(ProcessorID());

    if passed then
        if memaddrdesc.memattrs.shareability != Shareability_NSH then
            passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    return passed;
```

Library pseudocode for aarch64/functions/exclusive/AArch64.IsExclusiveVA

```
// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address only.
boolean AArch64.IsExclusiveVA(bits(64) address, integer processorid, integer size);
```

Library pseudocode for aarch64/functions/exclusive/AArch64.MarkExclusiveVA

```
// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.
AArch64.MarkExclusiveVA(bits(64) address, integer processorid, integer size);
```

Library pseudocode for aarch64/functions/exclusive/AArch64.SetExclusiveMonitors

```
// AArch64.SetExclusiveMonitors()
// ==============================
// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

AArch64.SetExclusiveMonitors(bits(64) address, integer size)
    acctype = AccType_ATOMIC;
    iswrite = FALSE;

    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);

    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;

    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    AArch64.MarkExclusiveVA(address, ProcessorID(), size);
```

Library pseudocode for aarch64/functions/fusedrstep/FPRSqrtStepFused

```
// FPRSqrtStepFused()
// ==================

bits(N) FPRSqrtStepFused(bits(N) op1_in, bits(N) op2)
    assert N IN {16, 32, 64};
    bits(N) result;
    bits(N) op1 = op1_in;
    boolean done;
    FPCRType fpcr = FPCR[];
    op1 = FPNeg(op1);
    boolean altfp = HaveAltFP() && fpcr.AH == '1';
    boolean fpexc = !altfp;                         // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11';             // Flush denormal input and output to zero
    if altfp then fpcr.RMode = '00';                // Use RNE rounding mode

    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);
    FPRounding rounding = FPRoundingMode(fpcr);

    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPOnePointFive('0', N);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, N);
        else
            // Fully fused multiply-add and halve
            result_value = (3.0 + (value1 * value2)) / 2.0;
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);

    return result;
```

Library pseudocode for aarch64/functions/fusedrstep/FPRecipStepFused

```
// FPRecipStepFused()
// ==================

bits(N) FPRecipStepFused(bits(N) op1_in, bits(N) op2)
    assert N IN {16, 32, 64};
    bits(N) op1 = op1_in;
    bits(N) result;
    boolean done;
    FPCRType fpcr = FPCR[];
    op1 = FPNeg(op1);

    boolean altfp = HaveAltFP() && fpcr.AH == '1';
    boolean fpexc = !altfp;                         // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11';             // Flush denormal input and output to zero
    if altfp then fpcr.RMode    = '00';             // Use RNE rounding mode

    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);
    FPRounding rounding = FPRoundingMode(fpcr);

    if !done then
        inf1  = (type1 == FPType_Infinity);
        inf2  = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo('0', N);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, N);
        else
            // Fully fused multiply-add
            result_value = 2.0 + (value1 * value2);
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);

    return result;
```

Library pseudocode for aarch64/functions/memory/AArch64.AccessIsTagChecked

```
// AArch64.AccessIsTagChecked()
// ============================
// TRUE if a given access is tag-checked, FALSE otherwise.

boolean AArch64.AccessIsTagChecked(bits(64) vaddr, AccType acctype)
    if PSTATE.M<4> == '1' then return FALSE;

    boolean is_instr = FALSE;
    if EffectiveTBI(vaddr, is_instr, PSTATE.EL) == '0' then
        return FALSE;

    if EffectiveTCMA(vaddr, PSTATE.EL) == '1' && (vaddr<59:55> == '00000' || vaddr<59:55> == '11111') the
        return FALSE;

    if !AArch64.AllocationTagAccessIsEnabled(acctype) then
        return FALSE;

    if acctype IN {AccType_IFETCH, AccType_TTW, AccType_DC, AccType_IC} then
        return FALSE;

    if acctype == AccType_NV2REGISTER then
        return FALSE;

    if PSTATE.TCO=='1' then
        return FALSE;

    if (HaveSME() && PSTATE.SM == '1' &&
        acctype IN {AccType_VEC, AccType_VECSTREAM,
                    AccType_SVE, AccType_SVESTREAM,
                    AccType_NONFAULT, AccType_CNOTFIRST} &&
        boolean IMPLEMENTATION_DEFINED "No tag checking of SIMD&FP loads and stores in Streaming SVE mode
        return FALSE;

    if (HaveSME() &&
        acctype IN {AccType_SME, AccType_SMESTREAM} &&
        boolean IMPLEMENTATION_DEFINED "No tag checking of SME LDR & STR instructions") then
        return FALSE;

    if !IsTagCheckedInstruction() then
        return FALSE;

    return TRUE;
```

Library pseudocode for aarch64/functions/memory/AArch64.AddressWithAllocationTag

```
// AArch64.AddressWithAllocationTag()
// ==================================
// Generate a 64-bit value containing a Logical Address Tag from a 64-bit
// virtual address and an Allocation Tag.
// If the extension is disabled, treats the Allocation Tag as '0000'.

bits(64) AArch64.AddressWithAllocationTag(bits(64) address, AccType acctype, bits(4) allocation_tag)
    bits(64) result = address;
    bits(4) tag;
    if AArch64.AllocationTagAccessIsEnabled(acctype) then
        tag = allocation_tag;
    else
        tag = '0000';
    result<59:56> = tag;
    return result;
```

Library pseudocode for aarch64/functions/memory/AArch64.AllocationTagFromAddress

```
// AArch64.AllocationTagFromAddress()
// ==================================
// Generate an Allocation Tag from a 64-bit value containing a Logical Address Tag.

bits(4) AArch64.AllocationTagFromAddress(bits(64) tagged_address)
    return tagged_address<59:56>;
```

Library pseudocode for aarch64/functions/memory/AArch64.CheckAlignment

```
// AArch64.CheckAlignment()
// ========================

boolean AArch64.CheckAlignment(bits(64) address, integer alignment, AccType acctype,
                               boolean iswrite)

    aligned = (address == Align(address, alignment));
    atomic  = acctype IN { AccType_ATOMIC, AccType_ATOMICRW, AccType_ORDEREDATOMIC,
                           AccType_ORDEREDATOMICRW, AccType_ATOMICLS64, AccType_A32LSMD};
    ordered = acctype IN { AccType_ORDERED, AccType_ORDEREDRW, AccType_LIMITEDORDERED,
                           AccType_ORDEREDATOMIC, AccType_ORDEREDATOMICRW };

    boolean check;
    if SCTLR[].A == '1' then check = TRUE;
    elsif HaveLSE2Ext() then
        check = (UInt(address<3:0>) + alignment > 16) && ((ordered && SCTLR[].nAA == '0') || atomic);
    else check = atomic || ordered;

    if check && !aligned then
        secondstage = FALSE;
        AArch64.Abort(address, AlignmentFault(acctype, iswrite, secondstage));

    return aligned;
```

Library pseudocode for aarch64/functions/memory/AArch64.CheckTag

```
// AArch64.CheckTag()
// ==================
// Performs a Tag Check operation for a memory access and returns
// whether the check passed

boolean AArch64.CheckTag(AddressDescriptor memaddrdesc, AccessDescriptor accdesc, bits(4) ptag, boolean w
    if memaddrdesc.memattrs.tagged then
        (memstatus, readtag) = PhysMemTagRead(memaddrdesc, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
        return ptag == readtag;
    else
        return TRUE;
```

Library pseudocode for aarch64/functions/memory/AArch64.MemSingle

```
// AArch64.MemSingle[] - non-assignment (read) form
// ================================================
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean aligned]
    boolean ispair = FALSE;
    return AArch64.MemSingle[address, size, acctype, aligned, ispair];

// AArch64.MemSingle[] - non-assignment (read) form
// ================================================
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean aligned, boolean
    assert size IN {1, 2, 4, 8, 16};
    constant halfsize = size DIV 2;
    if HaveLSE2Ext() then
        assert CheckAllInAlignedQuantity(address, size, 16);
    else
        assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    bits(size*8) value;
    iswrite = FALSE;

    boolean istagaccess = HaveMTE2Ext() && AArch64.AccessIsTagChecked(ZeroExtend(address, 64),
                                                                      acctype);
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Memory array access
    AccessDescriptor accdesc;
    if HaveTME() then
        accdesc = CreateAccessDescriptor(acctype);
        accdesc.transactional = TSTATE.depth > 0 && !(acctype IN {AccType_IFETCH,AccType_TTW});
        if accdesc.transactional && !MemHasTransactionalAccess(memaddrdesc.mematttrs) then
            FailTransaction(TMFailure_IMP, FALSE);
    else
        accdesc = CreateAccessDescriptor(acctype);

    boolean istagchecked = istagaccess;
    if istagaccess then
        bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
            AArch64.TagCheckFault(ZeroExtend(address, 64), acctype, iswrite);

    (atomic, splitpair) = CheckSingleAccessAttributes(address, memaddrdesc.mematttrs, size, acctype, iswri
    PhysMemRetStatus memstatus;
    if atomic then
        (memstatus, value) = PhysMemRead(memaddrdesc, size, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, size, accdesc);
    elsif splitpair then
        assert ispair;
        bits(halfsize * 8) lowhalf, highhalf;
        (memstatus, lowhalf) = PhysMemRead(memaddrdesc, halfsize, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, halfsize, accdesc);
        memaddrdesc.paddress.address = memaddrdesc.paddress.address + halfsize;
        (memstatus, highhalf) = PhysMemRead(memaddrdesc, halfsize, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, halfsize, accdesc);

        value = highhalf:lowhalf;
    else
        for i = 0 to size-1
            (memstatus, value<8*i+7:8*i>) = PhysMemRead(memaddrdesc, 1, accdesc);
            if IsFault(memstatus) then
                HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
```

```
            memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
    return value;

// AArch64.MemSingle[] - assignment (write) form
// =============================================

AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean aligned] = bits(size*8) value
    boolean ispair = FALSE;
    AArch64.MemSingle[address, size, acctype, aligned, ispair] = value;
    return;

// AArch64.MemSingle[] - assignment (write) form
// =============================================
// Perform an atomic, little-endian write of 'size' bytes.

AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean aligned, boolean ispair] = bit
    assert size IN {1, 2, 4, 8, 16};
    constant halfsize = size DIV 2;
    if HaveLSE2Ext() then
        assert CheckAllInAlignedQuantity(address, size, 16);
    else
        assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    iswrite = TRUE;

    boolean istagaccess = HaveMTE2Ext() && AArch64.AccessIsTagChecked(ZeroExtend(address, 64),
                                                                      acctype);
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.mematttrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    // Memory array access
    AccessDescriptor accdesc;
    if HaveTME() then
        accdesc = CreateAccessDescriptor(acctype);
        accdesc.transactional = TSTATE.depth > 0;
        if accdesc.transactional && !MemHasTransactionalAccess(memaddrdesc.mematttrs) then
            FailTransaction(TMFailure_IMP, FALSE);
    else
        accdesc = CreateAccessDescriptor(acctype);

    boolean istagchecked = istagaccess;
    if istagchecked then
        bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
            AArch64.TagCheckFault(ZeroExtend(address, 64), acctype, iswrite);

    PhysMemRetStatus memstatus;
    (atomic, splitpair) = CheckSingleAccessAttributes(address, memaddrdesc.mematttrs, size, acctype, iswri
    if atomic then
        memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);
    elsif splitpair then
        assert ispair;
        bits(halfsize*8) lowhalf, highhalf;
        <highhalf, lowhalf> = value;

        memstatus = PhysMemWrite(memaddrdesc, halfsize, accdesc, lowhalf);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, halfsize, accdesc);
        memaddrdesc.paddress.address = memaddrdesc.paddress.address + halfsize;
        memstatus = PhysMemWrite(memaddrdesc, halfsize, accdesc, highhalf);
        if IsFault(memstatus) then
```

```
        HandleExternalWriteAbort(memstatus, memaddrdesc, halfsize, accdesc);
else
    for i = 0 to size-1
        memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, value<8*i+7:8*i>);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
        memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
return;
```

Library pseudocode for aarch64/functions/memory/AArch64.MemTag

```
// AArch64.MemTag[] - non-assignment (read) form
// =============================================
// Load an Allocation Tag from memory.

bits(4) AArch64.MemTag[bits(64) address, AccType acctype]
    AddressDescriptor memaddrdesc;
    bits(4) value;

    iswrite = FALSE;
    aligned = TRUE;
    istagaccess = AArch64.AllocationTagAccessIsEnabled(acctype);
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned,
                                           TAG_GRANULE);
    accdesc = CreateAccessDescriptor(acctype);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Return the granule tag if tagging is enabled...
    if istagaccess && memaddrdesc.mematttrs.tagged then
        (memstatus, tag) = PhysMemTagRead(memaddrdesc, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
        return tag;
    else
        // ...otherwise read tag as zero.
        return '0000';

// AArch64.MemTag[] - assignment (write) form
// ==========================================
// Store an Allocation Tag to memory.

AArch64.MemTag[bits(64) address, AccType acctype] = bits(4) value
    AddressDescriptor memaddrdesc;
    iswrite = TRUE;

    // Stores of allocation tags must be aligned
    if address != Align(address, TAG_GRANULE) then
        boolean secondstage = FALSE;
        AArch64.Abort(address, AlignmentFault(acctype, iswrite, secondstage));

    aligned = TRUE;
    istagaccess = AArch64.AllocationTagAccessIsEnabled(acctype);
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned,
                                           TAG_GRANULE);

    // It is CONSTRAINED UNPREDICTABLE if tags stored to memory locations marked as Device
    // generate an Alignment Fault or store the data to locations.
    if memaddrdesc.mematttrs.memtype == MemType_Device then
        c = ConstrainUnpredictable(Unpredictable_DEVICETAGSTORE);
        assert c IN {Constraint_NONE, Constraint_FAULT};
        if c == Constraint_FAULT then
            boolean secondstage = FALSE;
            AArch64.Abort(address, AlignmentFault(acctype, iswrite, secondstage));

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    accdesc = CreateAccessDescriptor(acctype);
    // Memory array access

    if istagaccess && memaddrdesc.mematttrs.tagged then
        memstatus = PhysMemTagWrite(memaddrdesc, accdesc, value);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
```

Library pseudocode for aarch64/functions/memory/AArch64.PhysicalTag

```
// AArch64.PhysicalTag()
// =====================
// Generate a Physical Tag from a Logical Tag in an address

bits(4) AArch64.PhysicalTag(bits(64) vaddr)
    return vaddr<59:56>;
```

Library pseudocode for aarch64/functions/memory/AArch64.TranslateAddressForAtomicAccess

```
// AArch64.TranslateAddressForAtomicAccess()
// =========================================
// Performs an alignment check for atomic memory operations.
// Also translates 64-bit Virtual Address into Physical Address.

AddressDescriptor AArch64.TranslateAddressForAtomicAccess(bits(64) address, integer sizeinbits)
    boolean iswrite = FALSE;
    size = sizeinbits DIV 8;

    assert size IN {1, 2, 4, 8, 16};

    aligned = AArch64.CheckAlignment(address, size, AccType_ATOMICRW, iswrite);

    // MMU or MPU lookup
    boolean istagaccess = HaveMTE2Ext() && AArch64.AccessIsTagChecked(address,
                                                                      AccType_ATOMICRW);
    memaddrdesc = AArch64.TranslateAddress(address, AccType_ATOMICRW, iswrite,
                                           aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.mematttrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    boolean istagchecked = istagaccess;
    if istagchecked then
        bits(4) ptag = AArch64.PhysicalTag(address);
        accdesc = CreateAccessDescriptor(AccType_ATOMICRW);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
            AArch64.TagCheckFault(address, AccType_ATOMICRW, iswrite);
    return memaddrdesc;
```

Library pseudocode for aarch64/functions/memory/AddressSupportsLS64

```
// Returns TRUE if the 64-byte block following the given address supports the
// LD64B and ST64B instructions, and FALSE otherwise.
boolean AddressSupportsLS64(bits(64) address);
```

Library pseudocode for aarch64/functions/memory/CheckAllInAlignedQuantity

```
// CheckAllInAlignedQuantity()
// ===========================
// Returns TRUE if all accessed bytes are within one aligned quantity, FALSE otherwise.

boolean CheckAllInAlignedQuantity(bits(64) address, integer size, integer alignment)
    assert(size <= alignment);
    return Align(address+size-1, alignment) == Align(address, alignment);
```

Library pseudocode for aarch64/functions/memory/CheckSPAlignment

```
// CheckSPAlignment()
// ==================
// Check correct stack pointer alignment for AArch64 state.

CheckSPAlignment()
    bits(64) sp = SP[];
    boolean stack_align_check;
    if PSTATE.EL == EL0 then
        stack_align_check = (SCTLR[].SA0 != '0');
    else
        stack_align_check = (SCTLR[].SA != '0');

    if stack_align_check && sp != Align(sp, 16) then
        AArch64.SPAlignmentFault();

    return;
```

Library pseudocode for aarch64/functions/memory/CheckSingleAccessAttributes

```
// CheckSingleAccessAttributes()
// =============================
//
// When FEAT_LSE2 is implemented, a MemSingle[] access needs to be further assessed once the memory
// attributes are determined.
// If it was aligned to access size or targets Normal Inner Write-Back, Outer Write-Back Cacheable
// memory then it is single copy atomic and there is no alignment fault.
// If not, for exclusives, atomics and non atomic acquire release instructions - it is CONSTRAINED UNPRED
// if they generate an alignment fault.  If they do not generate an alignement fault - they are
// single copy atomic.
// Otherwise it is IMPLEMENTATION DEFINED - if they are single copy atomic.
//
// The function returns (atomic, splitpair), where
//     atomic indicates if the access is single copy atomic.
//     splitpair indicates that a load/store pair is split into 2 single copy atomic accesses.
//     when atomic and splitpair are both FALSE - the access is not single copy atomic and may be treated
//        as byte accesses.

(boolean, boolean) CheckSingleAccessAttributes(bits(64) address, MemoryAttributes memattrs, integer size,
                        AccType acctype, boolean iswrite, boolean aligned, boolean ispair)
    isnormalwb = (memattrs.memtype     == MemType_Normal &&
                  memattrs.inner.attrs == MemAttr_WB &&
                  memattrs.outer.attrs == MemAttr_WB);

    atomic    = TRUE;
    splitpair = FALSE;
    if isnormalwb then return (atomic, splitpair);

    accatomic  = acctype IN { AccType_ATOMIC, AccType_ATOMICRW, AccType_ORDEREDATOMIC,
                 AccType_ORDEREDATOMICRW, AccType_ATOMICLS64, AccType_A32LSMD};
    ordered = acctype IN { AccType_ORDERED, AccType_ORDEREDRW, AccType_LIMITEDORDERED, AccType_ORDEREDATO

    if !aligned && (accatomic || ordered) then
        atomic = ConstrainUnpredictableBool(Unpredictable_MISALIGNEDATOMIC);
        if !atomic then
            secondstage = FALSE;
            AArch64.Abort(address, AlignmentFault(acctype, iswrite, secondstage));
        else
            return (atomic, splitpair);

    if ispair && aligned then
        // load / store pair requests that are aligned to each register access are split into 2 single co
        atomic    = FALSE;
        splitpair = TRUE;
        return (atomic, splitpair);

    if aligned then
        return (atomic, splitpair);

    atomic = boolean IMPLEMENTATION_DEFINED "Misaligned accesses within 16 byte aligned memory but not No

    return (atomic, splitpair);
```

Library pseudocode for aarch64/functions/memory/IsTagCheckedInstruction

```
// Returns True if the current instruction uses tag-checked memory access,
// False otherwise.
boolean IsTagCheckedInstruction();
```

Library pseudocode for aarch64/functions/memory/Mem

```
// Mem[] - non-assignment (read) form
// ==================================
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch64.MemSingle directly.

bits(size*8) Mem[bits(64) address, integer size, AccType acctype]
    boolean ispair = FALSE;
    return Mem[address, size, acctype, ispair];

bits(size*8) Mem[bits(64) address, integer size, AccType acctype, boolean ispair]
    assert size IN {1, 2, 4, 8, 16};
    constant halfsize = size DIV 2;
    bits(size * 8) value;
    bits(halfsize * 8) lowhalf, highhalf;
    boolean iswrite = FALSE;
    boolean aligned;
    if ispair then
        // check alignment on size of element accessed, not overall access size
        aligned = AArch64.CheckAlignment(address, halfsize, acctype, iswrite);
    else
        aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    boolean atomic;

    if size != 16 || !(acctype IN {AccType_VEC, AccType_VECSTREAM}) then
        if !HaveLSE2Ext() then
            atomic = aligned;
        else
            atomic = CheckAllInAlignedQuantity(address, size, 16);

    elsif acctype IN {AccType_VEC, AccType_VECSTREAM} then
        // 128-bit SIMD&FP loads are treated as a pair of 64-bit single-copy atomic accesses
        // 64-bit aligned.
        atomic = address == Align(address, 8);
    else
        // 16-byte integer access
        atomic = address == Align(address, 16);

    if !atomic && ispair && address == Align(address, halfsize) then
        single_is_pair = FALSE;
        single_is_aligned = TRUE;
        lowhalf  = AArch64.MemSingle[address, halfsize, acctype,
                                     single_is_aligned, single_is_pair];
        highhalf = AArch64.MemSingle[address + halfsize, halfsize, acctype,
                                     single_is_aligned, single_is_pair];

        value = highhalf:lowhalf;
    elsif atomic && ispair then
        value = AArch64.MemSingle[address, size, acctype, aligned, ispair];
    elsif !atomic then

        assert size > 1;
        value<7:0> = AArch64.MemSingle[address, 1, acctype, aligned];

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        if !aligned then
            c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
            assert c IN {Constraint_FAULT, Constraint_NONE};
            if c == Constraint_NONE then aligned = TRUE;
        for i = 1 to size-1
            value<8*i+7:8*i> = AArch64.MemSingle[address+i, 1, acctype, aligned];

    elsif size == 16 && acctype IN {AccType_VEC, AccType_VECSTREAM} then
        lowhalf = AArch64.MemSingle[address, halfsize, acctype, aligned, ispair];
        highhalf = AArch64.MemSingle[address + halfsize, halfsize, acctype, aligned, ispair];
        value = highhalf:lowhalf;
    else
        value = AArch64.MemSingle[address, size, acctype, aligned, ispair];
```

```
    if BigEndian(acctype) then
        value = BigEndianReverse(value);

    return value;

// Mem[] - assignment (write) form
// ===============================
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem[bits(64) address, integer size, AccType acctype] = bits(size*8) value_in
    boolean ispair = FALSE;
    Mem[address, size, acctype, ispair] = value_in;

Mem[bits(64) address, integer size, AccType acctype, boolean ispair] = bits(size*8) value_in
    boolean iswrite = TRUE;
    constant halfsize = size DIV 2;
    bits(size*8) value = value_in;
    bits(halfsize*8) lowhalf, highhalf;
    boolean atomic;
    boolean aligned;
    if BigEndian(acctype) then
        value = BigEndianReverse(value);

    if ispair then
        // check alignment on size of element accessed, not overall access size
        aligned = AArch64.CheckAlignment(address, halfsize, acctype, iswrite);
    else
        aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    if ispair then
        atomic = CheckAllInAlignedQuantity(address, size, 16);

    elsif size != 16 || !(acctype IN {AccType_VEC, AccType_VECSTREAM}) then
        if !HaveLSE2Ext() then
            atomic = aligned;
        else
            atomic = CheckAllInAlignedQuantity(address, size, 16);
    elsif (acctype IN {AccType_VEC, AccType_VECSTREAM}) then
        // 128-bit SIMD&FP stores are treated as a pair of 64-bit single-copy atomic accesses
        // 64-bit aligned.
        atomic = address == Align(address, 8);
    else
        // 16-byte integer access
        atomic = address == Align(address, 16);

    if !atomic && ispair && address == Align(address, halfsize) then
        single_is_aligned = TRUE;
        <highhalf, lowhalf> = value;
        AArch64.MemSingle[address, halfsize, acctype,
                          single_is_aligned, ispair] = lowhalf;
        AArch64.MemSingle[address + halfsize, halfsize, acctype,
                          single_is_aligned, ispair] = highhalf;
    elsif atomic && ispair then
        AArch64.MemSingle[address, size, acctype, aligned, ispair] = value;
    elsif !atomic then
        assert size > 1;
        AArch64.MemSingle[address, 1, acctype, aligned] = value<7:0>;

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        if !aligned then
            c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
            assert c IN {Constraint_FAULT, Constraint_NONE};
            if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            AArch64.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
    elsif size == 16 && acctype IN {AccType_VEC, AccType_VECSTREAM} then
        <highhalf, lowhalf> = value;
        AArch64.MemSingle[address, halfsize, acctype, aligned, ispair] = lowhalf;
```

```
        AArch64.MemSingle[address + halfsize, halfsize, acctype, aligned, ispair] = highhalf;
    else
        AArch64.MemSingle[address, size, acctype, aligned, ispair] = value;
    return;
```

Library pseudocode for aarch64/functions/memory/MemAtomic

```
// MemAtomic()
// ===========
// Performs load and store memory operations for a given virtual address.

bits(size) MemAtomic(bits(64) address, MemAtomicOp op, bits(size) value, AccType ldacctype, AccType stacc
    bits(size) newvalue;
    memaddrdesc = AArch64.TranslateAddressForAtomicAccess(address, size);
    ldaccdesc = CreateAccessDescriptor(ldacctype);
    staccdesc = CreateAccessDescriptor(stacctype);

    // All observers in the shareability domain observe the
    // following load and store atomically.
    (memstatus, oldvalue) = PhysMemRead(memaddrdesc, size DIV 8, ldaccdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, size DIV 8, ldaccdesc);
    if BigEndian(ldacctype) then
        oldvalue = BigEndianReverse(oldvalue);

    case op of
        when MemAtomicOp_ADD   newvalue = oldvalue + value;
        when MemAtomicOp_BIC   newvalue = oldvalue AND NOT(value);
        when MemAtomicOp_EOR   newvalue = oldvalue EOR value;
        when MemAtomicOp_ORR   newvalue = oldvalue OR value;
        when MemAtomicOp_SMAX  newvalue = if SInt(oldvalue) > SInt(value) then oldvalue else value;
        when MemAtomicOp_SMIN  newvalue = if SInt(oldvalue) > SInt(value) then value else oldvalue;
        when MemAtomicOp_UMAX  newvalue = if UInt(oldvalue) > UInt(value) then oldvalue else value;
        when MemAtomicOp_UMIN  newvalue = if UInt(oldvalue) > UInt(value) then value else oldvalue;
        when MemAtomicOp_SWP   newvalue = value;

    if BigEndian(stacctype) then
        newvalue = BigEndianReverse(newvalue);
    memstatus = PhysMemWrite(memaddrdesc, size DIV 8, staccdesc, newvalue);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, size DIV 8, staccdesc);

    // Load operations return the old (pre-operation) value
    return oldvalue;
```

Library pseudocode for aarch64/functions/memory/MemAtomicCompareAndSwap

```
// MemAtomicCompareAndSwap()
// =========================
// Compares the value stored at the passed-in memory address against the passed-in expected
// value. If the comparison is successful, the value at the passed-in memory address is swapped
// with the passed-in new_value.

bits(size) MemAtomicCompareAndSwap(bits(64) address, bits(size) expectedvalue,
                                   bits(size) newvalue_in, AccType ldacctype, AccType stacctype)
    bits(size) newvalue = newvalue_in;
    memaddrdesc = AArch64.TranslateAddressForAtomicAccess(address, size);
    ldaccdesc = CreateAccessDescriptor(ldacctype);
    staccdesc = CreateAccessDescriptor(stacctype);

    // All observers in the shareability domain observe the
    // following load and store atomically.
    (memstatus, oldvalue) = PhysMemRead(memaddrdesc, size DIV 8, ldaccdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, size DIV 8, ldaccdesc);
    if BigEndian(ldacctype) then
        oldvalue = BigEndianReverse(oldvalue);

    if oldvalue == expectedvalue then
        if BigEndian(stacctype) then
            newvalue = BigEndianReverse(newvalue);
        memstatus = PhysMemWrite(memaddrdesc, size DIV 8, staccdesc, newvalue);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, size DIV 8, staccdesc);

    return oldvalue;
```

Library pseudocode for aarch64/functions/memory/MemLoad64B

```
// MemLoad64B()
// ============
// Performs an atomic 64-byte read from a given virtual address.

bits(512) MemLoad64B(bits(64) address, AccType acctype)
    bits(512) data;
    boolean iswrite = FALSE;
    constant integer size = 64;

    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);

    if !AddressSupportsLS64(address) then
        c = ConstrainUnpredictable(Unpredictable_LS64UNSUPPORTED);
        assert c IN {Constraint_LIMITED_ATOMICITY, Constraint_FAULT};

        if c == Constraint_FAULT then
            // Generate a stage 1 Data Abort reported using the DFSC code of 110101.
            boolean secondstage = FALSE;
            boolean s2fs1walk = FALSE;
            FaultRecord fault = AArch64.ExclusiveFault(acctype, iswrite, secondstage, s2fs1walk);
            AArch64.Abort(address, fault);
        else
            // Accesses are not single-copy atomic above the byte level
            for i = 0 to 63
                data<7+8*i : 8*i> = AArch64.MemSingle[address+8*i, 1, acctype, aligned];
            return data;

    AddressDescriptor memaddrdesc;
    istagaccess = HaveMTE2Ext() && AArch64.AccessIsTagChecked(address, acctype);
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);
    if HaveTME() then
        accdesc.transactional = TSTATE.depth > 0;

    boolean istagchecked = istagaccess;
    if istagchecked then
        bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
            AArch64.TagCheckFault(address, acctype, iswrite);

    PhysMemRetStatus memstatus;
    (memstatus, data) = PhysMemRead(memaddrdesc, size, accdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, size, accdesc);
    return data;
```

Library pseudocode for aarch64/functions/memory/MemStore64B

```
// MemStore64B()
// =============
// Performs an atomic 64-byte store to a given virtual address. Function does
// not return the status of the store.

MemStore64B(bits(64) address, bits(512) value, AccType acctype)
    boolean iswrite = TRUE;
    constant integer size = 64;
    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);

    if !AddressSupportsLS64(address) then
        c = ConstrainUnpredictable(Unpredictable_LS64UNSUPPORTED);
        assert c IN {Constraint_LIMITED_ATOMICITY, Constraint_FAULT};

        if c == Constraint_FAULT then
            // Generate a Data Abort reported using the DFSC code of 110101.
            boolean secondstage = FALSE;
            boolean s2fs1walk = FALSE;
            fault = AArch64.ExclusiveFault(acctype, iswrite, secondstage, s2fs1walk);
            AArch64.Abort(address, fault);
        else
            // Accesses are not single-copy atomic above the byte level.
            for i = 0 to 63
                AArch64.MemSingle[address+8*i, 1, acctype, aligned] = value<7+8*i : 8*i>;
    else
        -= MemStore64BWithRet(address, value, acctype);  // Return status is ignored by ST64B
    return;
```

Library pseudocode for aarch64/functions/memory/MemStore64BWithRet

```
// MemStore64BWithRet()
// ====================
// Performs an atomic 64-byte store to a given virtual address returning
// the status value of the operation.

bits(64) MemStore64BWithRet(bits(64) address, bits(512) value, AccType acctype)
    AddressDescriptor memaddrdesc;
    boolean iswrite = TRUE;
    constant integer size = 64;

    aligned     = AArch64.CheckAlignment(address, size, acctype, iswrite);
    istagaccess = HaveMTE2Ext() && AArch64.AccessIsTagChecked(address, acctype);
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);
        return ZeroExtend('1', 64);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), 64);

    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);
    if HaveTME() then
        accdesc.transactional = TSTATE.depth > 0;

    boolean istagchecked = istagaccess;
    if istagchecked then
        bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
            AArch64.TagCheckFault(address, acctype, iswrite);
            return ZeroExtend('1', 64);

    memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);
    return memstatus.store64bstatus;
```

Library pseudocode for aarch64/functions/memory/MemStore64BWithRetStatus

```
// Generates the return status of memory write with ST64BV or ST64BV0
// instructions. The status indicates if the operation succeeded, failed,
// or was not supported at this memory location.
bits(64) MemStore64BWithRetStatus();
```

Library pseudocode for aarch64/functions/memory/NVMem

```
// NVMem[] - non-assignment form
// ============================
// This function is the load memory access for the transformed System register read access
// when Enhanced Nested Virtualization is enabled with HCR_EL2.NV2 = 1.
// The address for the load memory access is calculated using
// the formula SignExtend(VNCR_EL2.BADDR : Offset<11:0>, 64) where,
//  * VNCR_EL2.BADDR holds the base address of the memory location, and
//  * Offset is the unique offset value defined architecturally for each System register that
//    supports transformation of register access to memory access.

bits(64) NVMem[integer offset]
    assert offset > 0;
    bits(64) address = SignExtend(VNCR_EL2.BADDR:offset<11:0>, 64);
    return Mem[address, 8, AccType_NV2REGISTER];

// NVMem[] - assignment form
// ============================
// This function is the store memory access for the transformed System register write access
// when Enhanced Nested Virtualization is enabled with HCR_EL2.NV2 = 1.
// The address for the store memory access is calculated using
// the formula SignExtend(VNCR_EL2.BADDR : Offset<11:0>, 64) where,
//  * VNCR_EL2.BADDR holds the base address of the memory location, and
//  * Offset is the unique offset value defined architecturally for each System register that
//    supports transformation of register access to memory access.

NVMem[integer offset] = bits(64) value
    assert offset > 0;
    bits(64) address = SignExtend(VNCR_EL2.BADDR:offset<11:0>, 64);
    Mem[address, 8, AccType_NV2REGISTER] = value;
    return;
```

Library pseudocode for aarch64/functions/memory/PhysMemTagRead

```
// This is the hardware operation which perform a single-copy atomic,
// Allocation Tag granule aligned, memory access from the tag in PA space.
//
// The function address the array using desc.paddress which supplies:
// * A 52-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of the array.
//
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,
// etc and other parameters required to access the physical memory or for setting syndrome
// register in the event of an External abort.
(PhysMemRetStatus, bits(4)) PhysMemTagRead(AddressDescriptor desc, AccessDescriptor accdesc);
```

Library pseudocode for aarch64/functions/memory/PhysMemTagWrite

```
// This is the hardware operation which perform a single-copy atomic,
// Allocation Tag granule aligned, memory access to the tag in PA space.
//
// The function address the array using desc.paddress which supplies:
// * A 52-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of the array.
//
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,
// etc and other parameters required to access the physical memory or for setting syndrome
// register in the event of an External abort.
PhysMemRetStatus PhysMemTagWrite(AddressDescriptor desc, AccessDescriptor accdesc, bits (4) value);
```

Library pseudocode for aarch64/functions/memory/SetTagCheckedInstruction

```
// Flag the current instruction as using/not using memory tag checking.
SetTagCheckedInstruction(boolean checked);
```

Library pseudocode for aarch64/functions/mops/CPYPostSizeChoice

```
// Returns the size of the copy that is performed by the CPYE* instructions for this
// implementation given the parameters of the destination, source and size of the copy.
// Postsize is encoded as -1*size for an option A implementation if cpysize is negative.
bits(64) CPYPostSizeChoice(bits(64) toaddress, bits(64) fromaddress, bits(64) cpysize);
```

Library pseudocode for aarch64/functions/mops/CPYPreSizeChoice

```
// Returns the size of the copy that is performed by the CPYP* instructions for this
// implementation given the parameters of the destination, source and size of the copy.
// Presize is encoded as -1*size for an option A implementation if cpysize is negative.
bits(64) CPYPreSizeChoice(bits(64) toaddress, bits(64) fromaddress, bits(64) cpysize);
```

Library pseudocode for aarch64/functions/mops/CPYSizeChoice

```
// Returns the size of the block this performed for an iteration of the copy given the
// parameters of the destination, source and size of the copy.
integer CPYSizeChoice(bits(64) toaddress, bits(64) fromaddress, bits(64) cpysize);
```

Library pseudocode for aarch64/functions/mops/CheckMOPSEnabled

```
// CheckMOPSEnabled()
// ==================
// Check for EL0 and EL1 access to the CPY* and SET* instructions.

CheckMOPSEnabled()
    if (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0') &&
        (!IsHCRXEL2Enabled() || HCRX_EL2.MSCEn == '0')) then
        UNDEFINED;

    if (PSTATE.EL == EL0 && SCTLR_EL1.MSCEn == '0' &&
        (!EL2Enabled() || HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0')) then
        UNDEFINED;

    if PSTATE.EL == EL0 && IsInHost() && SCTLR_EL2.MSCEn == '0' then
        UNDEFINED;
```

Library pseudocode for aarch64/functions/mops/MOPSStage

```
enumeration MOPSStage { MOPSStage_Prologue, MOPSStage_Main, MOPSStage_Epilogue };
```

Library pseudocode for aarch64/functions/mops/MaxBlockSizeCopiedBytes

```
// MaxBlockSizeCopiedBytes()
// =========================
// Returns the maximum number of bytes that can used in a single block of the copy.

integer MaxBlockSizeCopiedBytes()
    return integer IMPLEMENTATION_DEFINED "Maximum bytes used in a single block of a copy";
```

Library pseudocode for aarch64/functions/mops/MemCpyAccessTypes

```
// MemCpyAccessTypes()
// ===================
// Return the read and write access types for a CPY* instruction.

(AccType, AccType) MemCpyAccessTypes(bits(4) options)
    unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled() &&
        HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
    unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

    runpriv_at_el1 = options<1> == '1' && unpriv_at_el1;
    runpriv_at_el2 = options<1> == '1' && unpriv_at_el2;
    wunpriv_at_el1 = options<0> == '1' && unpriv_at_el1;
    wunpriv_at_el2 = options<0> == '1' && unpriv_at_el2;

    user_access_override = HaveUAOExt() && PSTATE.UAO == '1';

    AccType racctype;
    if !user_access_override && (runpriv_at_el1 || runpriv_at_el2) then
        racctype = if options<3> == '0' then AccType_UNPRIV else AccType_UNPRIVSTREAM;
    else
        racctype = if options<3> == '0' then AccType_NORMAL else AccType_STREAM;

    AccType wacctype;
    if !user_access_override && (wunpriv_at_el1 || wunpriv_at_el2) then
        wacctype = if options<2> == '0' then AccType_UNPRIV else AccType_UNPRIVSTREAM;
    else
        wacctype = if options<2> == '0' then AccType_NORMAL else AccType_STREAM;

    return (racctype, wacctype);
```

Library pseudocode for aarch64/functions/mops/MemCpyDirectionChoice

```
// Returns true if in the non-overlapping case of a memcpy of size cpysize bytes
// from the source address fromaddress to destination address toaddress is done
// in the forward direction on this implementation.
boolean MemCpyDirectionChoice(bits(64) fromaddress, bits(64) toaddress, bits(64) cpysize);
```

Library pseudocode for aarch64/functions/mops/MemCpyOptionA

```
// MemCpyOptionA()
// ===============
// Returns TRUE if the implementation uses Option A for the
// CPY*/SET* instructions, and FALSE otherwise.

boolean MemCpyOptionA()
    return boolean IMPLEMENTATION_DEFINED "CPY*/SET* instructions use Option A";
```

Library pseudocode for aarch64/functions/mops/MemCpyParametersIllformedE

```
// Returns TRUE if the inputs are not well formed (in terms of their size and/or alignment)
// for a CPYE* instruction for this implementation given the parameters of the destination,
// source and size of the copy.
boolean MemCpyParametersIllformedE(bits(64) toaddress, bits(64) fromaddress,
                                   bits(64) cpysize);
```

Library pseudocode for aarch64/functions/mops/MemCpyParametersIllformedM

```
// Returns TRUE if the inputs are not well formed (in terms of their size and/or alignment)
// for a CPYM* instruction for this implementation given the parameters of the destination,
// source and size of the copy.
boolean MemCpyParametersIllformedM(bits(64) toaddress, bits(64) fromaddress,
                                   bits(64) cpysize);
```

Library pseudocode for aarch64/functions/mops/MemCpyZeroSizeCheck

```
// Returns TRUE if the implementation option is checked on a copy of size zero remaining.
boolean MemCpyZeroSizeCheck();
```

Library pseudocode for aarch64/functions/mops/MemSetAccessType

```
// MemSetAccessType()
// ==================
// Return the access type for a SET* instruction.

AccType MemSetAccessType(bits(2) options)
    unpriv_at_el1 = options<0> == '1' && PSTATE.EL == EL1 && !(EL2Enabled() &&
        HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
    unpriv_at_el2 = (options<0> == '1' && PSTATE.EL == EL2 &&
        HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11');

    user_access_override = HaveUAOExt() && PSTATE.UAO == '1';

    AccType acctype;
    if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
        acctype = if options<1> == '0' then AccType_UNPRIV else AccType_UNPRIVSTREAM;
    else
        acctype = if options<1> == '0' then AccType_NORMAL else AccType_STREAM;

    return acctype;
```

Library pseudocode for aarch64/functions/mops/MemSetParametersIllformedE

```
// Returns TRUE if the inputs are not well formed (in terms of their size and/or
// alignment) for a SETE* or SETGE* instruction for this implementation given the
// parameters of the destination and size of the set.
boolean MemSetParametersIllformedE(bits(64) toaddress, bits(64) setsize,
                                   boolean IsSETGE);
```

Library pseudocode for aarch64/functions/mops/MemSetParametersIllformedM

```
// Returns TRUE if the inputs are not well formed (in terms of their size and/or
// alignment) for a SETM* or SETGM* instruction for this implementation given the
// parameters of the destination and size of the copy.
boolean MemSetParametersIllformedM(bits(64) toaddress, bits(64) setsize,
                                   boolean IsSETGM);
```

Library pseudocode for aarch64/functions/mops/MemSetZeroSizeCheck

```
// Returns TRUE if the implementation option is checked on a copy of size zero remaining.
boolean MemSetZeroSizeCheck();
```

Library pseudocode for aarch64/functions/mops/MismatchedCpySetTargetEL

```
// MismatchedCpySetTargetEL()
// ==========================
// Return the target exception level for an Exception_MemCpyMemSet.

bits(2) MismatchedCpySetTargetEL()
    bits(2) target_el;

    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;
    elsif (PSTATE.EL == EL1 && EL2Enabled() &&
        IsHCRXEL2Enabled() && HCRX_EL2.MCE2 == '1') then
        target_el = EL2;
    else
        target_el = EL1;

    return target_el;
```

Library pseudocode for aarch64/functions/mops/MismatchedMemCpyException

```
// MismatchedMemCpyException()
// ===========================
// Generates an exception for a CPY* instruction if the version
// is inconsistent with the state of the call.

MismatchedMemCpyException(boolean option_a, integer destreg, integer srcreg, integer sizereg,
                          boolean wrong_option, boolean from_epilogue, bits(4) options)
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    integer vect_offset = 0x0;
    bits(2) target_el = MismatchedCpySetTargetEL();

    ExceptionRecord exception = ExceptionSyndrome(Exception_MemCpyMemSet);
    exception.syndrome<24>    = '0';
    exception.syndrome<23>    = '0';
    exception.syndrome<22:19> = options;
    exception.syndrome<18>    = if from_epilogue then '1' else '0';
    exception.syndrome<17>    = if wrong_option then '1' else '0';
    exception.syndrome<16>    = if option_a then '1' else '0';
    // exception.syndrome<15> is RES0
    exception.syndrome<14:10> = destreg<4:0>;
    exception.syndrome<9:5>   = srcreg<4:0>;
    exception.syndrome<4:0>   = sizereg<4:0>;

    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/functions/mops/MismatchedMemSetException

```
// MismatchedMemSetException()
// ===========================
// Generates an exception for a SET* instruction if the version
// is inconsistent with the state of the call.

MismatchedMemSetException(boolean option_a, integer destreg, integer datareg, integer sizereg,
                          boolean wrong_option, boolean from_epilogue, bits(2) options,
                          boolean is_SETG)
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    integer vect_offset = 0x0;
    bits(2) target_el = MismatchedCpySetTargetEL();

    ExceptionRecord exception = ExceptionSyndrome(Exception_MemCpyMemSet);
    exception.syndrome<24>    = '1';
    exception.syndrome<23>    = if is_SETG then '1' else '0';
    // exception.syndrome<22:21> is RES0
    exception.syndrome<20:19> = options;
    exception.syndrome<18>    = if from_epilogue then '1' else '0';
    exception.syndrome<17>    = if wrong_option then '1' else '0';
    exception.syndrome<16>    = if option_a then '1' else '0';
    // exception.syndrome<15> is RES0
    exception.syndrome<14:10> = destreg<4:0>;
    exception.syndrome<9:5>   = datareg<4:0>;
    exception.syndrome<4:0>   = sizereg<4:0>;

    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/functions/mops/SETPostSizeChoice

```
// Returns the size of the set that is performed by the SETE* or SETGE* instructions
// for this implementation, given the parameters of the destination and size of the set.
// Postsize is encoded as -1*size for an option A implementation if setsize is negative.
bits(64) SETPostSizeChoice(bits(64) toaddress, bits(64) setsize, boolean IsSETGE);
```

Library pseudocode for aarch64/functions/mops/SETPreSizeChoice

```
// Returns the size of the set that is performed by the SETP* or SETGP* instructions
// for this implementation, given the parameters of the destination and size of the set.
// Presize is encoded as -1*size for an option A implementation if setsize is negative.
bits(64) SETPreSizeChoice(bits(64) toaddress, bits(64) setsize, boolean IsSETGP);
```

Library pseudocode for aarch64/functions/mops/SETSizeChoice

```
// Returns the size of the block thisperformed for an iteration of the set given
// the parameters of the destination and size of the set. The size of the block
// is an integer multiple of AlignSize.
integer SETSizeChoice(bits(64) toaddress, bits(64) setsize, integer AlignSize);
```

Library pseudocode for aarch64/functions/pac/addpac/AddPAC

```
// AddPAC()
// ========
// Calculates the pointer authentication code for a 64-bit quantity and then
// inserts that into pointer authentication code field of that 64-bit quantity.

bits(64) AddPAC(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data)
    bits(64) PAC;
    bits(64) result;
    bits(64) ext_ptr;
    bits(64) extfield;
    bit selbit;
    boolean auth = TRUE;
    boolean tbi = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';
    integer top_bit = if tbi then 55 else 63;

    // If tagged pointers are in use for a regime with two TTBRs, use bit<55> of
    // the pointer to select between upper and lower ranges, and preserve this.
    // This handles the awkward case where there is apparently no correct choice between
    // the upper and lower address range - ie an addr of 1xxxxxx0... with TBI0=0 and TBI1=1
    // and 0xxxxxx1 with TBI1=0 and TBI0=1:
    if PtrHasUpperAndLowerAddRanges() then
        assert S1TranslationRegime() IN {EL1, EL2};
        if S1TranslationRegime() == EL1 then
            // EL1 translation regime registers
            if data then
                if TCR_EL1.TBI1 == '1' || TCR_EL1.TBI0 == '1' then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            else
                if ((TCR_EL1.TBI1 == '1' && TCR_EL1.TBID1 == '0') ||
                    (TCR_EL1.TBI0 == '1' && TCR_EL1.TBID0 == '0')) then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
        else
            // EL2 translation regime registers
            if data then
                if TCR_EL2.TBI1 == '1' || TCR_EL2.TBI0 == '1' then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
            else
                if ((TCR_EL2.TBI1 == '1' && TCR_EL2.TBID1 == '0') ||
                    (TCR_EL2.TBI0 == '1' && TCR_EL2.TBID0 == '0')) then
                    selbit = ptr<55>;
                else
                    selbit = ptr<63>;
    else selbit = if tbi then ptr<55> else ptr<63>;

    if HaveEnhancedPAC2() && ConstPACField() then selbit = ptr<55>;
    integer bottom_PAC_bit = CalculateBottomPACBit(selbit);

    extfield = Replicate(selbit, 64);

    // Compute the pointer authentication code for a ptr with good extension bits
    if tbi then
        ext_ptr = (ptr<63:56> :
                   extfield<(56-bottom_PAC_bit)-1:0> : ptr<bottom_PAC_bit-1:0>);
    else
        ext_ptr =  extfield<(64-bottom_PAC_bit)-1:0> : ptr<bottom_PAC_bit-1:0>;

    PAC = ComputePAC(ext_ptr, modifier, K<127:64>, K<63:0>, auth);

    // Check if the ptr has good extension bits and corrupt the pointer authentication code if not
    if !IsZero(ptr<top_bit:bottom_PAC_bit>) && !IsOnes(ptr<top_bit:bottom_PAC_bit>) then
        if HaveEnhancedPAC() then
            PAC = 0x0000000000000000<63:0>;
        elsif !HaveEnhancedPAC2() then
            PAC<top_bit-1> = NOT(PAC<top_bit-1>);
```

```
    // Preserve the determination between upper and lower address at bit<55> and insert PAC into
    // bits that are not used for the address or the tag(s).
    if !HaveEnhancedPAC2() then
        if tbi then
            result = ptr<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
        else
            result = PAC<63:56>:selbit:PAC<54:bottom_PAC_bit>:ptr<bottom_PAC_bit-1:0>;
    else
        if tbi then
            result = (ptr<63:56>                                 : selbit :
                      (ptr<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>) :
                      ptr<bottom_PAC_bit-1:0>);
        else
            result = ((ptr<63:56> EOR PAC<63:56>)                : selbit :
                      (ptr<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>) :
                      ptr<bottom_PAC_bit-1:0>);
    return result;
```

Library pseudocode for aarch64/functions/pac/addpacda/AddPACDA

```
// AddPACDA()
// ==========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y and the
// APDAKey_EL1.

bits(64) AddPACDA(bits(64) x, bits(64) y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APDAKey_EL1;

    APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnDA else SCTLR_EL2.EnDA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                       (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnDA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return x;
    elsif TrapEL3 && EL3SDDTrapPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return AddPAC(x, y, APDAKey_EL1, TRUE);
```

Library pseudocode for aarch64/functions/pac/addpacdb/AddPACDB

```
// AddPACDB()
// ==========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y and the
// APDBKey_EL1.

bits(64) AddPACDB(bits(64) x, bits(64) y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APDBKey_EL1;

    APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnDB else SCTLR_EL2.EnDB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                       (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnDB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return x;
    elsif TrapEL3 && EL3SDDTrapPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return AddPAC(x, y, APDBKey_EL1, TRUE);
```

Library pseudocode for aarch64/functions/pac/addpacga/AddPACGA

```
// AddPACGA()
// ==========
// Returns a 64-bit value where the lower 32 bits are 0, and the upper 32 bits contain
// a 32-bit pointer authentication code which is derived using a cryptographic
// algorithm as a combination of x, y and the APGAKey_EL1.

bits(64) AddPACGA(bits(64) x, bits(64) y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(128) APGAKey_EL1;
    boolean auth = FALSE;

    APGAKey_EL1 = APGAKeyHi_EL1<63:0> : APGAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                       (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if TrapEL3 && EL3SDDTrapPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return ComputePAC(x, y, APGAKey_EL1<127:64>, APGAKey_EL1<63:0>, auth)<63:32>:Zeros(32);
```

Library pseudocode for aarch64/functions/pac/addpacia/AddPACIA

```
// AddPACIA()
// ==========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y, and the
// APIAKey_EL1.

bits(64) AddPACIA(bits(64) x, bits(64) y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APIAKey_EL1;

    APIAKey_EL1 = APIAKeyHi_EL1<63:0>:APIAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnIA else SCTLR_EL2.EnIA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                       (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return x;
    elsif TrapEL3 && EL3SDDTrapPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return AddPAC(x, y, APIAKey_EL1, FALSE);
```

Library pseudocode for aarch64/functions/pac/addpacib/AddPACIB

```
// AddPACIB()
// ==========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with a pointer authentication code, where the pointer authentication
// code is derived using a cryptographic algorithm as a combination of x, y and the
// APIBKey_EL1.

bits(64) AddPACIB(bits(64) x, bits(64) y)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APIBKey_EL1;

    APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnIB else SCTLR_EL2.EnIB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                       (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return x;
    elsif TrapEL3 && EL3SDDTrapPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return AddPAC(x, y, APIBKey_EL1, FALSE);
```

Library pseudocode for aarch64/functions/pac/auth/AArch64.PACFailException

```
// AArch64.PACFailException()
// ==========================
// Generates a PAC Fail Exception

AArch64.PACFailException(bits(2) syndrome)
    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_PACFail);
    exception.syndrome<1:0>   = syndrome;
    exception.syndrome<24:2>  = Zeros(23);               // RES0

    if UInt(PSTATE.EL) > UInt(EL0) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/functions/pac/auth/Auth

```
// Auth()
// ======
// Restores the upper bits of the address to be all zeros or all ones (based on the
// value of bit[55]) and computes and checks the pointer authentication code. If the
// check passes, then the restored address is returned. If the check fails, the
// second-top and third-top bits of the extension bits in the pointer authentication code
// field are corrupted to ensure that accessing the address will give a translation fault.

bits(64) Auth(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data, bit key_number,
              boolean is_combined)
    bits(64) PAC;
    bits(64) result;
    bits(64) original_ptr;
    bits(2) error_code;
    bits(64) extfield;
    boolean auth = TRUE;

    // Reconstruct the extension field used of adding the PAC to the pointer
    boolean tbi = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';
    integer bottom_PAC_bit = CalculateBottomPACBit(ptr<55>);
    extfield = Replicate(ptr<55>, 64);

    if tbi then
        original_ptr = (ptr<63:56> :
                        extfield<(56-bottom_PAC_bit)-1:0> : ptr<bottom_PAC_bit-1:0>);
    else
        original_ptr =  extfield<(64-bottom_PAC_bit)-1:0> : ptr<bottom_PAC_bit-1:0>;

    PAC = ComputePAC(original_ptr, modifier, K<127:64>, K<63:0>, auth);
    // Check pointer authentication code
    if tbi then
        if !HaveEnhancedPAC2() then
            if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> then
                result = original_ptr;
            else
                error_code = key_number:NOT(key_number);
                result = original_ptr<63:55>:error_code:original_ptr<52:0>;
        else
            result = ptr;
            result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
            if HaveFPACCombined() || (HaveFPAC() && !is_combined) then
                if result<54:bottom_PAC_bit> != Replicate(result<55>, (55-bottom_PAC_bit)) then
                    error_code = (if data then '1' else '0'):key_number;
                    AArch64.PACFailException(error_code);
    else
        if !HaveEnhancedPAC2() then
            if PAC<54:bottom_PAC_bit> == ptr<54:bottom_PAC_bit> && PAC<63:56> == ptr<63:56> then
                result = original_ptr;
            else
                error_code = key_number:NOT(key_number);
                result = original_ptr<63>:error_code:original_ptr<60:0>;
        else
            result = ptr;
            result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
            result<63:56> = result<63:56> EOR PAC<63:56>;
            if HaveFPACCombined() || (HaveFPAC() && !is_combined) then
                if result<63:bottom_PAC_bit> != Replicate(result<55>, (64-bottom_PAC_bit)) then
                    error_code = (if data then '1' else '0'):key_number;
                    AArch64.PACFailException(error_code);
    return result;
```

Library pseudocode for aarch64/functions/pac/authda/AuthDA

```
// AuthDA()
// ========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of x, using the same
// algorithm and key as AddPACDA().

bits(64) AuthDA(bits(64) x, bits(64) y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APDAKey_EL1;

    APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnDA else SCTLR_EL2.EnDA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                       (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnDA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return x;
    elsif TrapEL3 && EL3SDDTrapPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return Auth(x, y, APDAKey_EL1, TRUE, '0', is_combined);
```

Library pseudocode for aarch64/functions/pac/authdb/AuthDB

```
// AuthDB()
// ========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a
// pointer authentication code in the pointer authentication code field bits of x, using
// the same algorithm and key as AddPACDB().

bits(64) AuthDB(bits(64) x, bits(64) y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APDBKey_EL1;

    APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnDB else SCTLR_EL2.EnDB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                       (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnDB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return x;
    elsif TrapEL3 && EL3SDDTrapPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return Auth(x, y, APDBKey_EL1, TRUE, '1', is_combined);
```

Library pseudocode for aarch64/functions/pac/authia/AuthIA

```
// AuthIA()
// ========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of x, using the same
// algorithm and key as AddPACIA().

bits(64) AuthIA(bits(64) x, bits(64) y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APIAKey_EL1;

    APIAKey_EL1 = APIAKeyHi_EL1<63:0> : APIAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnIA else SCTLR_EL2.EnIA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                       (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return x;
    elsif TrapEL3 && EL3SDDTrapPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return Auth(x, y, APIAKey_EL1, FALSE, '0', is_combined);
```

Library pseudocode for aarch64/functions/pac/authib/AuthIB

```
// AuthIB()
// ========
// Returns a 64-bit value containing x, but replacing the pointer authentication code
// field bits with the extension of the address bits. The instruction checks a pointer
// authentication code in the pointer authentication code field bits of x, using the same
// algorithm and key as AddPACIB().

bits(64) AuthIB(bits(64) x, bits(64) y, boolean is_combined)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1)  Enable;
    bits(128) APIBKey_EL1;

    APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnIB else SCTLR_EL2.EnIB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                       (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return x;
    elsif TrapEL3 && EL3SDDTrapPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return Auth(x, y, APIBKey_EL1, FALSE, '1', is_combined);
```

Library pseudocode for aarch64/functions/pac/calcbottompacbit/CalculateBottomPACBit

```
// CalculateBottomPACBit()
// =======================

integer CalculateBottomPACBit(bit top_bit)
    integer tsz_field;
    boolean using64k;
    Constraint c;

    if PtrHasUpperAndLowerAddRanges() then
        assert S1TranslationRegime() IN {EL1, EL2};
        if S1TranslationRegime() == EL1 then
            // EL1 translation regime registers
            tsz_field = if top_bit == '1' then UInt(TCR_EL1.T1SZ) else UInt(TCR_EL1.T0SZ);
            using64k = if top_bit == '1' then TCR_EL1.TG1 == '11' else TCR_EL1.TG0 == '01';
        else
            // EL2 translation regime registers
            assert HaveEL(EL2);
            tsz_field = if top_bit == '1' then UInt(TCR_EL2.T1SZ) else UInt(TCR_EL2.T0SZ);
            using64k = if top_bit == '1' then TCR_EL2.TG1 == '11' else TCR_EL2.TG0 == '01';
    else
        tsz_field = if PSTATE.EL == EL2 then UInt(TCR_EL2.T0SZ) else UInt(TCR_EL3.T0SZ);
        using64k = if PSTATE.EL == EL2 then TCR_EL2.TG0 == '01' else TCR_EL3.TG0 == '01';

    max_limit_tsz_field = (if !HaveSmallTranslationTableExt() then 39 else if using64k then 47 else 48);
    if tsz_field > max_limit_tsz_field then
        // TCR_ELx.TySZ is out of range
        c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
        assert c IN {Constraint_FORCE, Constraint_NONE};
        if c == Constraint_FORCE then tsz_field = max_limit_tsz_field;
    tszmin = if using64k && AArch64.VAMax() == 52 then 12 else 16;
    if tsz_field < tszmin then
        c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
        assert c IN {Constraint_FORCE, Constraint_NONE};
        if c == Constraint_FORCE then tsz_field = tszmin;
    return (64-tsz_field);
```

Library pseudocode for aarch64/functions/pac/computepac/ComputePAC

```
// ComputePAC()
// ============

bits(64) ComputePAC(bits(64) data, bits(64) modifier, bits(64) key0, bits(64) key1, boolean auth)
    if UsePACIMP(auth) then
        return ComputePACIMPDEF(data, modifier, key0, key1);
    if UsePACQARMA3(auth) then
        boolean isqarma3 = TRUE;
        return ComputePACQARMA(data, modifier, key0, key1, isqarma3);
    if UsePACQARMA5(auth) then
        boolean isqarma3 = FALSE;
        return ComputePACQARMA(data, modifier, key0, key1, isqarma3);
```

Library pseudocode for aarch64/functions/pac/computepac/ComputePACIMPDEF

```
// Compute IMPLEMENTATION DEFINED cryptographic algorithm to be used for PAC calculation.
bits(64) ComputePACIMPDEF(bits(64) data, bits(64) modifier, bits(64) key0, bits(64) key1);
```

Library pseudocode for aarch64/functions/pac/computepac/ComputePACQARMA

```
// ComputePACQARMA()
// =================
// Compute QARMA3 or QARMA5 cryptographic algorithm for PAC calculation

bits(64) ComputePACQARMA(bits(64) data, bits(64) modifier, bits(64) key0,
                         bits(64) key1, boolean isqarma3)
    bits(64)  workingval;
    bits(64)  runningmod;
    bits(64)  roundkey;
    bits(64)  modk0;
    constant bits(64) Alpha = 0xC0AC29B7C97C50DD<63:0>;

    integer iterations;
    RC[0] = 0x0000000000000000<63:0>;
    RC[1] = 0x13198A2E03707344<63:0>;
    RC[2] = 0xA4093822299F31D0<63:0>;

    if isqarma3 then
        iterations = 2;
    else // QARMA5
        iterations = 4;
        RC[3] = 0x082EFA98EC4E6C89<63:0>;
        RC[4] = 0x452821E638D01377<63:0>;

    modk0 = key0<0>:key0<63:2>:(key0<63> EOR key0<1>);
    runningmod = modifier;
    workingval = data EOR key0;

    for i = 0 to iterations
        roundkey = key1 EOR runningmod;
        workingval  = workingval EOR roundkey;
        workingval = workingval EOR RC[i];
        if i > 0 then
            workingval = PACCellShuffle(workingval);
            workingval = PACMult(workingval);
        if isqarma3 then
            workingval = PACSub1(workingval);
        else
            workingval = PACSub(workingval);
        runningmod = TweakShuffle(runningmod<63:0>);
    roundkey = modk0 EOR runningmod;
    workingval = workingval EOR roundkey;
    workingval = PACCellShuffle(workingval);
    workingval = PACMult(workingval);
    if isqarma3 then
        workingval = PACSub1(workingval);
    else
        workingval = PACSub(workingval);
    workingval = PACCellShuffle(workingval);
    workingval = PACMult(workingval);
    workingval = key1 EOR workingval;
    workingval = PACCellInvShuffle(workingval);
    if isqarma3 then
        workingval = PACSub1(workingval);
    else
        workingval = PACInvSub(workingval);
    workingval = PACMult(workingval);
    workingval = PACCellInvShuffle(workingval);
    workingval = workingval EOR key0;
    workingval = workingval EOR runningmod;
    for i = 0 to iterations
        if isqarma3 then
            workingval = PACSub1(workingval);
        else
            workingval = PACInvSub(workingval);
        if i < iterations then
            workingval = PACMult(workingval);
            workingval = PACCellInvShuffle(workingval);
        runningmod = TweakInvShuffle(runningmod<63:0>);
        roundkey = key1 EOR runningmod;
```

```
        workingval = workingval EOR RC[iterations-i];
        workingval = workingval EOR roundkey;
        workingval = workingval EOR Alpha;
    workingval = workingval EOR modk0;

    return workingval;
```

Library pseudocode for aarch64/functions/pac/computepac/PACCellInvShuffle

```
// PACCellInvShuffle()
// ===================

bits(64) PACCellInvShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<15:12>;
    outdata<7:4> = indata<27:24>;
    outdata<11:8> = indata<51:48>;
    outdata<15:12> = indata<39:36>;
    outdata<19:16> = indata<59:56>;
    outdata<23:20> = indata<47:44>;
    outdata<27:24> = indata<7:4>;
    outdata<31:28> = indata<19:16>;
    outdata<35:32> = indata<35:32>;
    outdata<39:36> = indata<55:52>;
    outdata<43:40> = indata<31:28>;
    outdata<47:44> = indata<11:8>;
    outdata<51:48> = indata<23:20>;
    outdata<55:52> = indata<3:0>;
    outdata<59:56> = indata<43:40>;
    outdata<63:60> = indata<63:60>;
    return outdata;
```

Library pseudocode for aarch64/functions/pac/computepac/PACCellShuffle

```
// PACCellShuffle()
// ================

bits(64) PACCellShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<55:52>;
    outdata<7:4> = indata<27:24>;
    outdata<11:8> = indata<47:44>;
    outdata<15:12> = indata<3:0>;
    outdata<19:16> = indata<31:28>;
    outdata<23:20> = indata<51:48>;
    outdata<27:24> = indata<7:4>;
    outdata<31:28> = indata<43:40>;
    outdata<35:32> = indata<35:32>;
    outdata<39:36> = indata<15:12>;
    outdata<43:40> = indata<59:56>;
    outdata<47:44> = indata<23:20>;
    outdata<51:48> = indata<11:8>;
    outdata<55:52> = indata<39:36>;
    outdata<59:56> = indata<19:16>;
    outdata<63:60> = indata<63:60>;
    return outdata;
```

Library pseudocode for aarch64/functions/pac/computepac/PACInvSub

```
// PACInvSub()
// ===========

bits(64) PACInvSub(bits(64) Tinput)
    // This is a 4-bit substitution from the PRINCE-family cipher
    bits(64) Toutput;
    for i = 0 to 15
        case Tinput<4*i+3:4*i> of
            when '0000'  Toutput<4*i+3:4*i> = '0101';
            when '0001'  Toutput<4*i+3:4*i> = '1110';
            when '0010'  Toutput<4*i+3:4*i> = '1101';
            when '0011'  Toutput<4*i+3:4*i> = '1000';
            when '0100'  Toutput<4*i+3:4*i> = '1010';
            when '0101'  Toutput<4*i+3:4*i> = '1011';
            when '0110'  Toutput<4*i+3:4*i> = '0001';
            when '0111'  Toutput<4*i+3:4*i> = '1001';
            when '1000'  Toutput<4*i+3:4*i> = '0010';
            when '1001'  Toutput<4*i+3:4*i> = '0110';
            when '1010'  Toutput<4*i+3:4*i> = '1111';
            when '1011'  Toutput<4*i+3:4*i> = '0000';
            when '1100'  Toutput<4*i+3:4*i> = '0100';
            when '1101'  Toutput<4*i+3:4*i> = '1100';
            when '1110'  Toutput<4*i+3:4*i> = '0111';
            when '1111'  Toutput<4*i+3:4*i> = '0011';
    return Toutput;
```

Library pseudocode for aarch64/functions/pac/computepac/PACMult

```
// PACMult()
// =========

bits(64) PACMult(bits(64) Sinput)
    bits(4)  t0;
    bits(4)  t1;
    bits(4)  t2;
    bits(4)  t3;
    bits(64) Soutput;

    for i = 0 to 3
        t0<3:0> = RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 2);
        t0<3:0> = t0<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
        t1<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
        t1<3:0> = t1<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 2);
        t2<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 2) EOR RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1);
        t2<3:0> = t2<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
        t3<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 2);
        t3<3:0> = t3<3:0> EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
        Soutput<4*i+3:4*i> = t3<3:0>;
        Soutput<4*(i+4)+3:4*(i+4)> = t2<3:0>;
        Soutput<4*(i+8)+3:4*(i+8)> = t1<3:0>;
        Soutput<4*(i+12)+3:4*(i+12)> = t0<3:0>;
    return Soutput;
```

Library pseudocode for aarch64/functions/pac/computepac/PACSub

```
// PACSub()
// ========

bits(64) PACSub(bits(64) Tinput)
    // This is a 4-bit substitution from the PRINCE-family cipher
    bits(64) Toutput;
    for i = 0 to 15
        case Tinput<4*i+3:4*i> of
            when '0000'  Toutput<4*i+3:4*i> = '1011';
            when '0001'  Toutput<4*i+3:4*i> = '0110';
            when '0010'  Toutput<4*i+3:4*i> = '1000';
            when '0011'  Toutput<4*i+3:4*i> = '1111';
            when '0100'  Toutput<4*i+3:4*i> = '1100';
            when '0101'  Toutput<4*i+3:4*i> = '0000';
            when '0110'  Toutput<4*i+3:4*i> = '1001';
            when '0111'  Toutput<4*i+3:4*i> = '1110';
            when '1000'  Toutput<4*i+3:4*i> = '0011';
            when '1001'  Toutput<4*i+3:4*i> = '0111';
            when '1010'  Toutput<4*i+3:4*i> = '0100';
            when '1011'  Toutput<4*i+3:4*i> = '0101';
            when '1100'  Toutput<4*i+3:4*i> = '1101';
            when '1101'  Toutput<4*i+3:4*i> = '0010';
            when '1110'  Toutput<4*i+3:4*i> = '0001';
            when '1111'  Toutput<4*i+3:4*i> = '1010';
    return Toutput;
```

Library pseudocode for aarch64/functions/pac/computepac/PacSub1

```
// PacSub1()
// =========

bits(64) PACSub1(bits(64) Tinput)
    // This is a 4-bit substitution from Qarma sigma1
    bits(64) Toutput;
    for i = 0 to 15
        case Tinput<4*i+3:4*i> of
            when '0000' Toutput<4*i+3:4*i> = '1010';
            when '0001' Toutput<4*i+3:4*i> = '1101';
            when '0010' Toutput<4*i+3:4*i> = '1110';
            when '0011' Toutput<4*i+3:4*i> = '0110';
            when '0100' Toutput<4*i+3:4*i> = '1111';
            when '0101' Toutput<4*i+3:4*i> = '0111';
            when '0110' Toutput<4*i+3:4*i> = '0011';
            when '0111' Toutput<4*i+3:4*i> = '0101';
            when '1000' Toutput<4*i+3:4*i> = '1001';
            when '1001' Toutput<4*i+3:4*i> = '1000';
            when '1010' Toutput<4*i+3:4*i> = '0000';
            when '1011' Toutput<4*i+3:4*i> = '1100';
            when '1100' Toutput<4*i+3:4*i> = '1011';
            when '1101' Toutput<4*i+3:4*i> = '0001';
            when '1110' Toutput<4*i+3:4*i> = '0010';
            when '1111' Toutput<4*i+3:4*i> = '0100';
    return Toutput;
```

Library pseudocode for aarch64/functions/pac/computepac/RC

```
array bits(64) RC[0..4];
```

Library pseudocode for aarch64/functions/pac/computepac/RotCell

```
// RotCell()
// =========

bits(4) RotCell(bits(4) incell, integer amount)
    bits(8) tmp;
    bits(4) outcell;

    // assert amount>3 || amount<1;
    tmp<7:0> = incell<3:0>:incell<3:0>;
    outcell = tmp<7-amount:4-amount>;
    return outcell;
```

Library pseudocode for aarch64/functions/pac/computepac/TweakCellInvRot

```
// TweakCellInvRot()
// =================

bits(4) TweakCellInvRot(bits(4) incell)
    bits(4) outcell;
    outcell<3> = incell<2>;
    outcell<2> = incell<1>;
    outcell<1> = incell<0>;
    outcell<0> = incell<0> EOR incell<3>;
    return outcell;
```

Library pseudocode for aarch64/functions/pac/computepac/TweakCellRot

```
// TweakCellRot()
// ==============

bits(4) TweakCellRot(bits(4) incell)
    bits(4) outcell;
    outcell<3> = incell<0> EOR incell<1>;
    outcell<2> = incell<3>;
    outcell<1> = incell<2>;
    outcell<0> = incell<1>;
    return outcell;
```

Library pseudocode for aarch64/functions/pac/computepac/TweakInvShuffle

```
// TweakInvShuffle()
// =================

bits(64) TweakInvShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = TweakCellInvRot(indata<51:48>);
    outdata<7:4> = indata<55:52>;
    outdata<11:8> = indata<23:20>;
    outdata<15:12> = indata<27:24>;
    outdata<19:16> = indata<3:0>;
    outdata<23:20> = indata<7:4>;
    outdata<27:24> = TweakCellInvRot(indata<11:8>);
    outdata<31:28> = indata<15:12>;
    outdata<35:32> = TweakCellInvRot(indata<31:28>);
    outdata<39:36> = TweakCellInvRot(indata<63:60>);
    outdata<43:40> = TweakCellInvRot(indata<59:56>);
    outdata<47:44> = TweakCellInvRot(indata<19:16>);
    outdata<51:48> = indata<35:32>;
    outdata<55:52> = indata<39:36>;
    outdata<59:56> = indata<43:40>;
    outdata<63:60> = TweakCellInvRot(indata<47:44>);
    return outdata;
```

Library pseudocode for aarch64/functions/pac/computepac/TweakShuffle

```
// TweakShuffle()
// ==============

bits(64) TweakShuffle(bits(64) indata)
    bits(64) outdata;
    outdata<3:0> = indata<19:16>;
    outdata<7:4> = indata<23:20>;
    outdata<11:8> = TweakCellRot(indata<27:24>);
    outdata<15:12> = indata<31:28>;
    outdata<19:16> = TweakCellRot(indata<47:44>);
    outdata<23:20> = indata<11:8>;
    outdata<27:24> = indata<15:12>;
    outdata<31:28> = TweakCellRot(indata<35:32>);
    outdata<35:32> = indata<51:48>;
    outdata<39:36> = indata<55:52>;
    outdata<43:40> = indata<59:56>;
    outdata<47:44> = TweakCellRot(indata<63:60>);
    outdata<51:48> = TweakCellRot(indata<3:0>);
    outdata<55:52> = indata<7:4>;
    outdata<59:56> = TweakCellRot(indata<43:40>);
    outdata<63:60> = TweakCellRot(indata<39:36>);
    return outdata;
```

Library pseudocode for aarch64/functions/pac/computepac/UsePACIMP

```
// UsePACIMP()
// ===========
// Checks whether IMPLEMENTATION DEFINED cryptographic algorithm to be used for PAC
// calculation.

boolean UsePACIMP(boolean auth)
    return if auth then HavePACIMPAuth() else HavePACIMPGeneric();
```

Library pseudocode for aarch64/functions/pac/computepac/UsePACQARMA3

```
// UsePACQARMA3()
// ==============
// Checks whether QARMA3 cryptographic algorithm to be used for PAC calculation.

boolean UsePACQARMA3(boolean auth)
    return if auth then HavePACQARMA3Auth() else HavePACQARMA3Generic();
```

Library pseudocode for aarch64/functions/pac/computepac/UsePACQARMA5

```
// UsePACQARMA5()
// ==============
// Checks whether QARMA5 cryptographic algorithm to be used for PAC calculation.

boolean UsePACQARMA5(boolean auth)
    return if auth then HavePACQARMA5Auth() else HavePACQARMA5Generic();
```

Library pseudocode for aarch64/functions/pac/pac/ConstPACField

```
// ConstPACField()
// ===============
// Returns TRUE if bit<55> can be used to determine the size of the PAC field, FALSE otherwise.

boolean ConstPACField()
    return (HaveEnhancedPAC2() &&
        boolean IMPLEMENTATION_DEFINED "Bit 55 determines the size of the PAC field");
```

Library pseudocode for aarch64/functions/pac/pac/HaveEnhancedPAC

```
// HaveEnhancedPAC()
// =================
// Returns TRUE if support for EnhancedPAC is implemented, FALSE otherwise.

boolean HaveEnhancedPAC()
    return (HavePACExt()
        && boolean IMPLEMENTATION_DEFINED "Has enhanced PAC functionality");
```

Library pseudocode for aarch64/functions/pac/pac/HaveEnhancedPAC2

```
// HaveEnhancedPAC2()
// ==================
// Returns TRUE if support for EnhancedPAC2 is implemented, FALSE otherwise.

boolean HaveEnhancedPAC2()
    return HasArchVersion(ARMv8p6) || (HasArchVersion(ARMv8p3)
            && boolean IMPLEMENTATION_DEFINED "Has enhanced PAC 2 functionality");
```

Library pseudocode for aarch64/functions/pac/pac/HaveFPAC

```
// HaveFPAC()
// ==========
// Returns TRUE if support for FPAC is implemented, FALSE otherwise.

boolean HaveFPAC()
    return HaveEnhancedPAC2() && boolean IMPLEMENTATION_DEFINED "Has FPAC functionality";
```

Library pseudocode for aarch64/functions/pac/pac/HaveFPACCombined

```
// HaveFPACCombined()
// ==================
// Returns TRUE if support for FPACCombined is implemented, FALSE otherwise.

boolean HaveFPACCombined()
    return HaveFPAC() && boolean IMPLEMENTATION_DEFINED "Has FPAC Combined functionality";
```

Library pseudocode for aarch64/functions/pac/pac/HavePACExt

```
// HavePACExt()
// ============
// Returns TRUE if support for the PAC extension is implemented, FALSE otherwise.

boolean HavePACExt()
    return HasArchVersion(ARMv8p3);
```

Library pseudocode for aarch64/functions/pac/pac/HavePACIMPAuth

```
// HavePACIMPAuth()
// ================
// Returns TRUE if support for PAC IMP Auth is implemented, FALSE otherwise.

boolean HavePACIMPAuth()
    return HavePACExt() && boolean IMPLEMENTATION_DEFINED "Has PAC IMPDEF Auth functionality";
```

Library pseudocode for aarch64/functions/pac/pac/HavePACIMPGeneric

```
// HavePACIMPGeneric()
// ===================
// Returns TRUE if support for PAC IMP Generic is implemented, FALSE otherwise.

boolean HavePACIMPGeneric()
    return HavePACExt() && boolean IMPLEMENTATION_DEFINED "Has PAC IMPDEF Generic functionality";
```

Library pseudocode for aarch64/functions/pac/pac/HavePACQARMA3Auth

```
// HavePACQARMA3Auth()
// ==================
// Returns TRUE if support for PAC QARMA3 Auth is implemented, FALSE otherwise.

boolean HavePACQARMA3Auth()
    return HavePACExt() && boolean IMPLEMENTATION_DEFINED "Has PAC QARMA3 Auth functionality";
```

Library pseudocode for aarch64/functions/pac/pac/HavePACQARMA3Generic

```
// HavePACQARMA3Generic()
// ======================
// Returns TRUE if support for PAC QARMA3 Generic is implemented, FALSE otherwise.

boolean HavePACQARMA3Generic()
    return HavePACExt() && boolean IMPLEMENTATION_DEFINED "Has PAC QARMA3 Generic functionality";
```

Library pseudocode for aarch64/functions/pac/pac/HavePACQARMA5Auth

```
// HavePACQARMA5Auth()
// ==================
// Returns TRUE if support for PAC QARMA5 Auth is implemented, FALSE otherwise.

boolean HavePACQARMA5Auth()
    return HavePACExt() && boolean IMPLEMENTATION_DEFINED "Has PAC QARMA5 Auth functionality";
```

Library pseudocode for aarch64/functions/pac/pac/HavePACQARMA5Generic

```
// HavePACQARMA5Generic()
// ======================
// Returns TRUE if support for PAC QARMA5 Generic is implemented, FALSE otherwise.

boolean HavePACQARMA5Generic()
    return HavePACExt() && boolean IMPLEMENTATION_DEFINED "Has PAC QARMA5 Generic functionality";
```

Library pseudocode for aarch64/functions/pac/pac/PtrHasUpperAndLowerAddRanges

```
// PtrHasUpperAndLowerAddRanges()
// ==============================
// Returns TRUE if the pointer has upper and lower address ranges, FALSE otherwise.

boolean PtrHasUpperAndLowerAddRanges()
    regime = TranslationRegime(PSTATE.EL, AccType_NORMAL);
    return HasUnprivileged(regime);
```

Library pseudocode for aarch64/functions/pac/strip/Strip

```
// Strip()
// =======
// Strip() returns a 64-bit value containing A, but replacing the pointer authentication
// code field bits with the extension of the address bits. This can apply to either
// instructions or data, where, as the use of tagged pointers is distinct, it might be
// handled differently.

bits(64) Strip(bits(64) A, boolean data)
    bits(64) original_ptr;
    bits(64) extfield;
    boolean tbi = EffectiveTBI(A, !data, PSTATE.EL) == '1';
    integer bottom_PAC_bit = CalculateBottomPACBit(A<55>);
    extfield = Replicate(A<55>, 64);

    if tbi then
        original_ptr = (A<63:56> :
                        extfield<(56-bottom_PAC_bit)-1:0> : A<bottom_PAC_bit-1:0>);
    else
        original_ptr =  extfield<(64-bottom_PAC_bit)-1:0> : A<bottom_PAC_bit-1:0>;

    return original_ptr;
```

Library pseudocode for aarch64/functions/pac/trappacuse/TrapPACUse

```
// TrapPACUse()
// ============
// Used for the trapping of the pointer authentication functions by higher exception
// levels.

TrapPACUse(bits(2) target_el)
    assert HaveEL(target_el) && target_el !=  EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr(64);
    ExceptionRecord exception;
    vect_offset = 0;
    exception = ExceptionSyndrome(Exception_PACTrap);
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/functions/ras/AArch64.ESBOperation

```
// AArch64.ESBOperation()
// ======================
// Perform the AArch64 ESB operation, either for ESB executed in AArch64 state, or for
// ESB in AArch32 state when SError interrupts are routed to an Exception level using
// AArch64

AArch64.ESBOperation()
    boolean mask_active;

    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1';
    route_to_el2 = (EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || HCR_EL2.AMO == '1'));

    target = if route_to_el3 then EL3 elsif route_to_el2 then EL2 else EL1;

    if target == EL1 then
        mask_active = PSTATE.EL IN {EL0, EL1};
    elsif HaveVirtHostExt() && target == EL2 && HCR_EL2.<E2H,TGE> == '11' then
        mask_active = PSTATE.EL IN {EL0, EL2};
    else
        mask_active = PSTATE.EL == target;

    mask_set = (PSTATE.A == '1' && (!HaveDoubleFaultExt() || SCR_EL3.EA == '0' ||
                                    PSTATE.EL != EL3 || SCR_EL3.NMEA == '0'));
    intdis = Halted() || ExternalDebugInterruptsDisabled(target);
    masked = (UInt(target) < UInt(PSTATE.EL)) || intdis || (mask_active && mask_set);

    // Check for a masked Physical SError pending that can be synchronized
    // by an Error synchronization event.
    if masked && IsSynchronizablePhysicalSErrorPending() then
        // This function might be called for an interworking case, and INTdis is masking
        // the SError interrupt.
        if ELUsingAArch32(S1TranslationRegime()) then
            syndrome32 = AArch32.PhysicalSErrorSyndrome();
            DISR = AArch32.ReportDeferredSError(syndrome32.AET, syndrome32.ExT);
        else
            implicit_esb = FALSE;
            syndrome64 = AArch64.PhysicalSErrorSyndrome(implicit_esb);
            DISR_EL1 = AArch64.ReportDeferredSError(syndrome64);
        ClearPendingPhysicalSError();               // Set ISR_EL1.A to 0

    return;
```

Library pseudocode for aarch64/functions/ras/AArch64.PhysicalSErrorSyndrome

```
// Return the SError syndrome
bits(25) AArch64.PhysicalSErrorSyndrome(boolean implicit_esb);
```

Library pseudocode for aarch64/functions/ras/AArch64.ReportDeferredSError

```
// AArch64.ReportDeferredSError()
// ==============================
// Generate deferred SError syndrome

bits(64) AArch64.ReportDeferredSError(bits(25) syndrome)
    bits(64) target;
    target<31>   = '1';              // A
    target<24>   = syndrome<24>;     // IDS
    target<23:0> = syndrome<23:0>;   // ISS
    return target;
```

Library pseudocode for aarch64/functions/ras/AArch64.vESBOperation

```
// AArch64.vESBOperation()
// =======================
// Perform the AArch64 ESB operation for virtual SError interrupts, either for ESB
// executed in AArch64 state, or for ESB in AArch32 state with EL2 using AArch64 state

AArch64.vESBOperation()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    // If physical SError interrupts are routed to EL2, and TGE is not set, then a virtual
    // SError interrupt might be pending
    vSEI_enabled = HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';
    vSEI_pending = vSEI_enabled && HCR_EL2.VSE == '1';
    vintdis      = Halted() || ExternalDebugInterruptsDisabled(EL1);
    vmasked      = vintdis || PSTATE.A == '1';

    // Check for a masked virtual SError pending
    if vSEI_pending && vmasked then
        // This function might be called for the interworking case, and INTdis is masking
        // the virtual SError interrupt.
        if ELUsingAArch32(EL1) then
            VDISR = AArch32.ReportDeferredSError(VDFSR<15:14>, VDFSR<12>);
        else
            VDISR_EL2 = AArch64.ReportDeferredSError(VSESR_EL2<24:0>);
        HCR_EL2.VSE = '0';                       // Clear pending virtual SError

    return;
```

Library pseudocode for aarch64/functions/registers/AArch64.MaybeZeroRegisterUppers

```
// AArch64.MaybeZeroRegisterUppers()
// =================================
// On taking an exception to  AArch64 from AArch32, it is CONSTRAINED UNPREDICTABLE whether the top
// 32 bits of registers visible at any lower Exception level using AArch32 are set to zero.

AArch64.MaybeZeroRegisterUppers()
    assert UsingAArch32();         // Always called from AArch32 state before entering AArch64 state

    integer first;
    integer last;
    boolean include_R15;
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
        first = 0;  last = 14;  include_R15 = FALSE;
    elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) then
        first = 0;  last = 30;  include_R15 = FALSE;
    else
        first = 0;  last = 30;  include_R15 = TRUE;

    for n = first to last
        if (n != 15 || include_R15) && ConstrainUnpredictableBool(Unpredictable_ZEROUPPER) then
            _R[n]<63:32> = Zeros(32);

    return;
```

Library pseudocode for aarch64/functions/registers/AArch64.ResetGeneralRegisters

```
// AArch64.ResetGeneralRegisters()
// ===============================

AArch64.ResetGeneralRegisters()

    for i = 0 to 30
        X[i, 64] = bits(64) UNKNOWN;

    return;
```

Library pseudocode for aarch64/functions/registers/AArch64.ResetSIMDFPRegisters

```
// AArch64.ResetSIMDFPRegisters()
// ==============================

AArch64.ResetSIMDFPRegisters()

    for i = 0 to 31
        V[i, 128] = bits(128) UNKNOWN;

    return;
```

Library pseudocode for aarch64/functions/registers/AArch64.ResetSpecialRegisters

```
// AArch64.ResetSpecialRegisters()
// ===============================

AArch64.ResetSpecialRegisters()

    // AArch64 special registers
    SP_EL0 = bits(64) UNKNOWN;
    SP_EL1 = bits(64) UNKNOWN;
    SPSR_EL1 = bits(64) UNKNOWN;
    ELR_EL1  = bits(64) UNKNOWN;
    if HaveEL(EL2) then
        SP_EL2 = bits(64) UNKNOWN;
        SPSR_EL2 = bits(64) UNKNOWN;
        ELR_EL2  = bits(64) UNKNOWN;
    if HaveEL(EL3) then
        SP_EL3 = bits(64) UNKNOWN;
        SPSR_EL3 = bits(64) UNKNOWN;
        ELR_EL3  = bits(64) UNKNOWN;

    // AArch32 special registers that are not architecturally mapped to AArch64 registers
    if HaveAArch32EL(EL1) then
        SPSR_fiq<31:0> = bits(32) UNKNOWN;
        SPSR_irq<31:0> = bits(32) UNKNOWN;
        SPSR_abt<31:0> = bits(32) UNKNOWN;
        SPSR_und<31:0> = bits(32) UNKNOWN;

    // External debug special registers
    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(64) UNKNOWN;

    return;
```

Library pseudocode for aarch64/functions/registers/AArch64.ResetSystemRegisters

```
AArch64.ResetSystemRegisters(boolean cold_reset);
```

Library pseudocode for aarch64/functions/registers/PC

```
// PC - non-assignment form
// ========================
// Read program counter.

bits(64) PC[]
    return _PC;
```

Library pseudocode for aarch64/functions/registers/SP

```
// SP[] - assignment form
// ======================
// Write to stack pointer from a 64-bit value.

SP[] = bits(64) value
    if PSTATE.SP == '0' then
        SP_EL0 = value;
    else
        case PSTATE.EL of
            when EL0  SP_EL0 = value;
            when EL1  SP_EL1 = value;
            when EL2  SP_EL2 = value;
            when EL3  SP_EL3 = value;
    return;

// SP[] - non-assignment form
// ==========================
// Read stack pointer with slice of 64 bits.

bits(64) SP[]
    if PSTATE.SP == '0' then
        return SP_EL0;
    else
        case PSTATE.EL of
            when EL0  return SP_EL0;
            when EL1  return SP_EL1;
            when EL2  return SP_EL2;
            when EL3  return SP_EL3;
```

Library pseudocode for aarch64/functions/registers/V

```
// V[] - assignment form
// =====================
// Write to SIMD&FP register with implicit extension from
// 8, 16, 32, 64 or 128 bits.

V[integer n, integer width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64,128};
    integer vlen = if IsSVEEnabled(PSTATE.EL) then CurrentVL else 128;
    if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
        _Z[n] = ZeroExtend(value, MAX_VL);
    else
        _Z[n]<vlen-1:0> = ZeroExtend(value, vlen);

// V[] - non-assignment form
// =========================
// Read from SIMD&FP register with implicit slice of 8, 16
// 32, 64 or 128 bits.

bits(width) V[integer n, integer width]
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64,128};
    return _Z[n]<width-1:0>;
```

Library pseudocode for aarch64/functions/registers/Vpart

```
// Vpart[] - non-assignment form
// ============================
// Reads a 128-bit SIMD&FP register in up to two parts:
//  part 0 returns the bottom 8, 16, 32 or 64 bits of a value held in the register;
//  part 1 returns the top half of the bottom 64 bits or the top half of the 128-bit
//  value held in the register.

bits(width) Vpart[integer n, integer part, integer width]
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
        assert width < 128;
        return V[n, width];
    else
        assert width IN {32,64};
        bits(128) vreg = V[n, 128];
        return vreg<(width * 2)-1:width>;

// Vpart[] - assignment form
// =========================
// Writes a 128-bit SIMD&FP register in up to two parts:
//  part 0 zero extends a 8, 16, 32, or 64-bit value to fill the whole register;
//  part 1 inserts a 64-bit value into the top half of the register.

Vpart[integer n, integer part, integer width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
        assert width < 128;
        V[n, width] = value;
    else
        assert width == 64;
        bits(64) vreg = V[n, 64];
        V[n, 128] = value<63:0> : vreg;
```

Library pseudocode for aarch64/functions/registers/X

```
// X[] - assignment form
// =====================
// Write to general-purpose register from either a 32-bit or a 64-bit value,
// where the size of the value is passed as an argument.

X[integer n, integer width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {32,64};
    if n != 31 then
        _R[n] = ZeroExtend(value, 64);
    return;

// X[] - non-assignment form
// =========================
// Read from general-purpose register with an explicit slice of 8, 16, 32 or 64 bits.

bits(width) X[integer n, integer width]
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64};
    if n != 31 then
        return _R[n]<width-1:0>;
    else
        return Zeros(width);
```

Library pseudocode for aarch64/functions/sme/HaveEBF16

```
// HaveEBF16()
// ===========
// Returns TRUE if the EBF16 extension is implemented, FALSE otherwise.

boolean HaveEBF16()
    return boolean IMPLEMENTATION_DEFINED "Have EBF16 extension";
```

Library pseudocode for aarch64/functions/sme/HaveSME

```
// HaveSME()
// =========
// Returns TRUE if the SME extension is implemented, FALSE otherwise.

boolean HaveSME()
    return boolean IMPLEMENTATION_DEFINED "Have SME extension";
```

Library pseudocode for aarch64/functions/sme/HaveSMEF64F64

```
// HaveSMEF64F64()
// ===============
// Returns TRUE if the SMEF64F64 extension is implemented, FALSE otherwise.

boolean HaveSMEF64F64()
    return HaveSME() && boolean IMPLEMENTATION_DEFINED "Have SMEF64F64 extension";
```

Library pseudocode for aarch64/functions/sme/HaveSMEI16I64

```
// HaveSMEI16I64()
// ===============
// Returns TRUE if the SMEI16I64 extension is implemented, FALSE otherwise.

boolean HaveSMEI16I64()
    return HaveSME() && boolean IMPLEMENTATION_DEFINED "Have SMEI16I64 extension";
```

Library pseudocode for aarch64/functions/sme/System

```
array bits(MAX_VL) _ZA[0..255];
```

Library pseudocode for aarch64/functions/sme/ZAhslice

```
// ZAhslice[] - non-assignment form
// ================================

bits(width) ZAhslice[integer tile, integer esize, integer slice, integer width]
    assert esize IN {8, 16, 32, 64, 128};
    integer tiles = esize DIV 8;
    assert tile >= 0 && tile < tiles;
    integer slices = SVL DIV esize;
    assert slice >= 0 && slice < slices;

    return ZAvector[tile + slice * tiles, width];

// ZAhslice[] - assignment form
// ============================

ZAhslice[integer tile, integer esize, integer slice, integer width] = bits(width) value
    assert esize IN {8, 16, 32, 64, 128};
    integer tiles = esize DIV 8;
    assert tile >= 0 && tile < tiles;
    integer slices = SVL DIV esize;
    assert slice >= 0 && slice < slices;

    ZAvector[tile + slice * tiles, width] = value;
```

Library pseudocode for aarch64/functions/sme/ZAslice

```
// ZAslice[] - non-assignment form
// ===============================

bits(width) ZAslice[integer tile, integer esize, boolean vertical, integer slice, integer width]
    bits(width) result;

    if vertical then
        result = ZAvslice[tile, esize, slice, width];
    else
        result = ZAhslice[tile, esize, slice, width];

    return result;

// ZAslice[] - assignment form
// ===========================

ZAslice[integer tile, integer esize, boolean vertical, integer slice, integer width] = bits(width) value
    if vertical then
        ZAvslice[tile, esize, slice, width] = value;
    else
        ZAhslice[tile, esize, slice, width] = value;
```

Library pseudocode for aarch64/functions/sme/ZAtile

```
// ZAtile[] - non-assignment form
// ==============================

bits(width) ZAtile[integer tile, integer esize, integer width]
    constant integer svl = SVL;
    integer slices = svl DIV esize;
    assert width == svl * slices;
    bits(width) result;

    for slice = 0 to slices-1
        Elem[result, slice, svl] = ZAhslice[tile, esize, slice, svl];

    return result;

// ZAtile[] - assignment form
// ==========================

ZAtile[integer tile, integer esize, integer width] = bits(width) value
    constant integer svl = SVL;
    integer slices = svl DIV esize;
    assert width == svl * slices;

    for slice = 0 to slices-1
        ZAhslice[tile, esize, slice, svl] = Elem[value, slice, svl];
```

Library pseudocode for aarch64/functions/sme/ZAvector

```
// ZAvector[] - non-assignment form
// ================================

bits(width) ZAvector[integer index, integer width]
    assert width == SVL;
    assert index >= 0 && index < (width DIV 8);

    return _ZA[index]<width-1:0>;

// ZAvector[] - assignment form
// ============================

ZAvector[integer index, integer width] = bits(width) value
    assert width == SVL;
    assert index >= 0 && index < (width DIV 8);

    if ConstrainUnpredictableBool(Unpredictable_SMEZEROUPPER) then
        _ZA[index] = ZeroExtend(value, MAX_VL);
    else
        _ZA[index]<width-1:0> = value;
```

Library pseudocode for aarch64/functions/sme/ZAvslice

```
// ZAvslice[] - non-assignment form
// ================================

bits(width) ZAvslice[integer tile, integer esize, integer slice, integer width]
    integer slices = SVL DIV esize;
    bits(width) result;

    for s = 0 to slices-1
        bits(width) hslice = ZAhslice[tile, esize, s, width];
        Elem[result, s, esize] = Elem[hslice, slice, esize];

    return result;

// ZAvslice[] - assignment form
// ============================

ZAvslice[integer tile, integer esize, integer slice, integer width] = bits(width) value
    integer slices = SVL DIV esize;

    for s = 0 to slices-1
        bits(width) hslice = ZAhslice[tile, esize, s, width];
        Elem[hslice, slice, esize] = Elem[value, s, esize];
        ZAhslice[tile, esize, s, width] = hslice;
```

Library pseudocode for aarch64/functions/sve/AArch32.IsFPEnabled

```
// AArch32.IsFPEnabled()
// =====================
// Returns TRUE if access to the SIMD&FP instructions or System registers are
// enabled at the target exception level in AArch32 state and FALSE otherwise.

boolean AArch32.IsFPEnabled(bits(2) el)
    if el == EL0 && !ELUsingAArch32(EL1) then
        return AArch64.IsFPEnabled(el);

    if HaveEL(EL3) && ELUsingAArch32(EL3) && CurrentSecurityState() == SS_NonSecure then
        // Check if access disabled in NSACR
        if NSACR.cp10 == '0' then return FALSE;

    if el IN {EL0, EL1} then
        // Check if access disabled in CPACR
        boolean disabled;
        case CPACR.cp10 of
            when '00' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '10' disabled = ConstrainUnpredictableBool(Unpredictable_RESCPACR);
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if !ELUsingAArch32(EL2) then
            return AArch64.IsFPEnabled(EL2);
        if HCPTR.TCP10 == '1' then return FALSE;

    if HaveEL(EL3) && !ELUsingAArch32(EL3) then
        // Check if access disabled in CPTR_EL3
        if CPTR_EL3.TFP == '1' then return FALSE;

    return TRUE;
```

Library pseudocode for aarch64/functions/sve/AArch64.IsFPEnabled

```
// AArch64.IsFPEnabled()
// =====================
// Returns TRUE if access to the SIMD&FP instructions or System registers are
// enabled at the target exception level in AArch64 state and FALSE otherwise.

boolean AArch64.IsFPEnabled(bits(2) el)
    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} && !IsInHost() then
        // Check FP&SIMD at EL0/EL1
        boolean disabled;
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    // Check if access disabled in CPTR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            boolean disabled;
            case CPTR_EL2.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = el == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then return FALSE;
        else
            if CPTR_EL2.TFP == '1' then return FALSE;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.TFP == '1' then return FALSE;

    return TRUE;
```

Library pseudocode for aarch64/functions/sve/AnyActiveElement

```
// AnyActiveElement()
// ==================
// Return TRUE if there is at least one active element in mask. Otherwise,
// return FALSE.

boolean AnyActiveElement(bits(N) mask, integer esize)
    return LastActiveElement(mask, esize) >= 0;
```

Library pseudocode for aarch64/functions/sve/BitDeposit

```
// BitDeposit()
// ============
// Deposit the least significant bits from DATA into result positions
// selected by non-zero bits in MASK, setting other result bits to zero.

bits(N) BitDeposit (bits(N) data, bits(N) mask)
    bits(N) res = Zeros(N);
    integer db = 0;
    for rb = 0 to N-1
        if mask<rb> == '1' then
            res<rb> = data<db>;
            db = db + 1;
    return res;
```

Library pseudocode for aarch64/functions/sve/BitExtract

```
// BitExtract()
// ============
// Extract and pack DATA bits selected by the non-zero bits in MASK into
// the least significant result bits, setting other result bits to zero.

bits(N) BitExtract (bits(N) data, bits(N) mask)
    bits(N) res = Zeros(N);
    integer rb = 0;
    for db = 0 to N-1
        if mask<db> == '1' then
            res<rb> = data<db>;
            rb = rb + 1;
    return res;
```

Library pseudocode for aarch64/functions/sve/BitGroup

```
// BitGroup()
// ==========
// Extract and pack DATA bits selected by the non-zero bits in MASK into
// the least significant result bits, and pack unselected bits into the
// most significant result bits.

bits(N) BitGroup (bits(N) data, bits(N) mask)
    bits(N) res;
    integer rb = 0;

    // compress masked bits to right
    for db = 0 to N-1
        if mask<db> == '1' then
            res<rb> = data<db>;
            rb = rb + 1;
    // compress unmasked bits to left
    for db = 0 to N-1
        if mask<db> == '0' then
            res<rb> = data<db>;
            rb = rb + 1;
    return res;
```

Library pseudocode for aarch64/functions/sve/CeilPow2

```
// CeilPow2()
// ==========
// For a positive integer X, return the smallest power of 2 >= X

integer CeilPow2(integer x)
    if x == 0 then return 0;
    if x == 1 then return 2;
    return FloorPow2(x - 1) * 2;
```

Library pseudocode for aarch64/functions/sve/CheckNonStreamingSVEEnabled

```
// CheckNonStreamingSVEEnabled()
// =============================
// Checks for traps on SVE instructions that are not legal in streaming mode.

CheckNonStreamingSVEEnabled()
    CheckSVEEnabled();

    if HaveSME() && PSTATE.SM == '1' && !IsFullA64Enabled() then
        SMEAccessTrap(SMEExceptionType_Streaming, PSTATE.EL);
```

Library pseudocode for aarch64/functions/sve/CheckNormalSVEEnabled

```
// CheckNormalSVEEnabled()
// =======================
// Checks for traps on normal SVE instructions and instructions that
// access SVE System registers.

CheckNormalSVEEnabled()
    boolean disabled;

    if (HaveEL(EL3) && (CPTR_EL3.EZ == '0' || CPTR_EL3.TFP == '1') &&
        EL3SDDTrapPriority()) then
        UNDEFINED;

    // Check if access disabled in CPACR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check SVE at EL0/EL1
        case CPACR_EL1.ZEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then SVEAccessTrap(EL1);

        // Check SIMD&FP at EL0/EL1
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

    // Check if access disabled in CPTR_EL2
    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            // Check SVE at EL2
            case CPTR_EL2.ZEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then SVEAccessTrap(EL2);

            // Check SIMD&FP at EL2
            case CPTR_EL2.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
        else
            if CPTR_EL2.TZ == '1' then SVEAccessTrap(EL2);
            if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.EZ == '0' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                SVEAccessTrap(EL3);

        if CPTR_EL3.TFP == '1' then
            if Halted() && EDSCR.SDD == '1' then
                UNDEFINED;
            else
                AArch64.AdvSIMDFPAccessTrap(EL3);
```

Library pseudocode for aarch64/functions/sve/CheckSMEAccess

```
// CheckSMEAccess()
// ================
// Check that access to SME System registers is enabled.

CheckSMEAccess()
    boolean disabled;
    // Check if access disabled in CPACR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check SME at EL0/EL1
        case CPACR_EL1.SMEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL1);

    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            // Check SME at EL2
            case CPTR_EL2.SMEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2);
        else
            if CPTR_EL2.TSM == '1' then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2);

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.ESM == '0' then SMEAccessTrap(SMEExceptionType_AccessTrap, EL3);
```

Library pseudocode for aarch64/functions/sve/CheckSMEAndZAEnabled

```
// CheckSMEAndZAEnabled()
// ======================

CheckSMEAndZAEnabled()
    CheckSMEEnabled();

    if PSTATE.ZA == '0' then
        SMEAccessTrap(SMEExceptionType_InactiveZA, PSTATE.EL);
```

Library pseudocode for aarch64/functions/sve/CheckSMEEnabled

```
// CheckSMEEnabled()
// =================

CheckSMEEnabled()
    boolean disabled;
    // Check if access disabled in CPACR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check SME at EL0/EL1
        case CPACR_EL1.SMEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL1);

        // Check SIMD&FP at EL0/EL1
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            // Check SME at EL2
            case CPTR_EL2.SMEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2);

            // Check SIMD&FP at EL2
            case CPTR_EL2.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
        else
            if CPTR_EL2.TSM == '1' then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2);
            if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.ESM == '0' then SMEAccessTrap(SMEExceptionType_AccessTrap, EL3);
        if CPTR_EL3.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL3);
```

Library pseudocode for aarch64/functions/sve/CheckSVEEnabled

```
// CheckSVEEnabled()
// =================
// Checks for traps on SVE instructions and instructions that
// access SVE System registers.

CheckSVEEnabled()
    if HaveSME() && (PSTATE.SM == '1' || !HaveSVE()) then
        CheckStreamingSVEEnabled();
    else
        CheckNormalSVEEnabled();
```

Library pseudocode for aarch64/functions/sve/CheckStreamingSVEAndZAEnabled

```
// CheckStreamingSVEAndZAEnabled()
// ===============================

CheckStreamingSVEAndZAEnabled()
    CheckStreamingSVEEnabled();

    if PSTATE.ZA == '0' then
        SMEAccessTrap(SMEExceptionType_InactiveZA, PSTATE.EL);
```

Library pseudocode for aarch64/functions/sve/CheckStreamingSVEEnabled

```
// CheckStreamingSVEEnabled()
// ==========================

CheckStreamingSVEEnabled()
    CheckSMEEnabled();

    if PSTATE.SM == '0' then
        SMEAccessTrap(SMEExceptionType_NotStreaming, PSTATE.EL);
```

Library pseudocode for aarch64/functions/sve/CurrentVL

```
// CurrentVL - non-assignment form
// ===============================

integer CurrentVL
    return if HaveSME() && PSTATE.SM == '1' then SVL else NVL;
```

Library pseudocode for aarch64/functions/sve/DecodePredCount

```
// DecodePredCount()
// =================

integer DecodePredCount(bits(5) pattern, integer esize)
    integer elements = CurrentVL DIV esize;
    integer numElem;
    case pattern of
        when '00000' numElem = FloorPow2(elements);
        when '00001' numElem = if elements >= 1 then 1 else 0;
        when '00010' numElem = if elements >= 2 then 2 else 0;
        when '00011' numElem = if elements >= 3 then 3 else 0;
        when '00100' numElem = if elements >= 4 then 4 else 0;
        when '00101' numElem = if elements >= 5 then 5 else 0;
        when '00110' numElem = if elements >= 6 then 6 else 0;
        when '00111' numElem = if elements >= 7 then 7 else 0;
        when '01000' numElem = if elements >= 8 then 8 else 0;
        when '01001' numElem = if elements >= 16 then 16 else 0;
        when '01010' numElem = if elements >= 32 then 32 else 0;
        when '01011' numElem = if elements >= 64 then 64 else 0;
        when '01100' numElem = if elements >= 128 then 128 else 0;
        when '01101' numElem = if elements >= 256 then 256 else 0;
        when '11101' numElem = elements - (elements MOD 4);
        when '11110' numElem = elements - (elements MOD 3);
        when '11111' numElem = elements;
        otherwise    numElem = 0;
    return numElem;
```

Library pseudocode for aarch64/functions/sve/ElemFFR

```
// ElemFFR[] - non-assignment form
// ===============================

bit ElemFFR[integer e, integer esize]
    return ElemP[_FFR, e, esize];

// ElemFFR[] - assignment form
// ===========================

ElemFFR[integer e, integer esize] = bit value
    integer psize = esize DIV 8;
    integer n = e * psize;
    assert n >= 0 && (n + psize) <= CurrentVL DIV 8;
    _FFR<n+psize-1:n> = ZeroExtend(value, psize);
    return;
```

Library pseudocode for aarch64/functions/sve/ElemP

```
// ElemP[] - non-assignment form
// =============================

bit ElemP[bits(N) pred, integer e, integer esize]
    assert esize IN {8, 16, 32, 64, 128};
    integer n = e * (esize DIV 8);
    assert n >= 0 && n < N;
    return pred<n>;

// ElemP[] - assignment form
// =========================

ElemP[bits(N) &pred, integer e, integer esize] = bit value
    assert esize IN {8, 16, 32, 64, 128};
    integer psize = esize DIV 8;
    integer n = e * psize;
    assert n >= 0 && (n + psize) <= N;
    pred<n+psize-1:n> = ZeroExtend(value, psize);
    return;
```

Library pseudocode for aarch64/functions/sve/FFR

```
// FFR[] - non-assignment form
// ===========================

bits(width) FFR[integer width]
    assert width == CurrentVL DIV 8;
    return _FFR<width-1:0>;

// FFR[] - assignment form
// =======================

FFR[integer width] = bits(width) value
    assert width == CurrentVL DIV 8;
    if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
        _FFR = ZeroExtend(value, MAX_PL);
    else
        _FFR<width-1:0> = value;
```

Library pseudocode for aarch64/functions/sve/FPCompareNE

```
// FPCompareNE()
// =============

boolean FPCompareNE(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    boolean result;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};
    op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};

    if op1_nan || op2_nan then
        result = TRUE;
        if type1 == FPType_SNaN || type2 == FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpcr);
    else // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 != value2);
        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for aarch64/functions/sve/FPCompareUN

```
// FPCompareUN()
// =============

boolean FPCompareUN(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    if type1 == FPType_SNaN || type2 == FPType_SNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);

    result = type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN};
    if !result then
        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for aarch64/functions/sve/FPConvertSVE

```
// FPConvertSVE()
// ==============

bits(M) FPConvertSVE(bits(N) op, FPCRType fpcr_in, FPRounding rounding, integer M)
    FPCRType fpcr = fpcr_in;
    fpcr.AHP = '0';
    return FPConvert(op, fpcr, rounding, M);

// FPConvertSVE()
// ==============

bits(M) FPConvertSVE(bits(N) op, FPCRType fpcr_in, integer M)
    FPCRType fpcr = fpcr_in;
    fpcr.AHP = '0';
    return FPConvert(op, fpcr, FPRoundingMode(fpcr), M);
```

Library pseudocode for aarch64/functions/sve/FPExpA

```
// FPExpA()
// ========

bits(N) FPExpA(bits(N) op)
    assert N IN {16,32,64};
    bits(N) result;
    bits(N) coeff;
    integer idx = if N == 16 then UInt(op<4:0>) else UInt(op<5:0>);
    coeff = FPExpCoefficient[idx, N];
    if N == 16 then
        result<15:0> = '0':op<9:5>:coeff<9:0>;
    elsif N == 32 then
        result<31:0> = '0':op<13:6>:coeff<22:0>;
    else // N == 64
        result<63:0> = '0':op<16:6>:coeff<51:0>;

    return result;
```

Library pseudocode for aarch64/functions/sve/FPExpCoefficient

```
// FPExpCoefficient()
// ==================

bits(N) FPExpCoefficient[integer index, integer N]
    assert N IN {16,32,64};
    integer result;

    if N == 16 then
        case index of
            when  0 result = 0x0000;
            when  1 result = 0x0016;
            when  2 result = 0x002d;
            when  3 result = 0x0045;
            when  4 result = 0x005d;
            when  5 result = 0x0075;
            when  6 result = 0x008e;
            when  7 result = 0x00a8;
            when  8 result = 0x00c2;
            when  9 result = 0x00dc;
            when 10 result = 0x00f8;
            when 11 result = 0x0114;
            when 12 result = 0x0130;
            when 13 result = 0x014d;
            when 14 result = 0x016b;
            when 15 result = 0x0189;
            when 16 result = 0x01a8;
            when 17 result = 0x01c8;
            when 18 result = 0x01e8;
            when 19 result = 0x0209;
            when 20 result = 0x022b;
            when 21 result = 0x024e;
            when 22 result = 0x0271;
            when 23 result = 0x0295;
            when 24 result = 0x02ba;
            when 25 result = 0x02e0;
            when 26 result = 0x0306;
            when 27 result = 0x032e;
            when 28 result = 0x0356;
            when 29 result = 0x037f;
            when 30 result = 0x03a9;
            when 31 result = 0x03d4;

    elsif N == 32 then
        case index of
            when  0 result = 0x000000;
            when  1 result = 0x0164d2;
            when  2 result = 0x02cd87;
            when  3 result = 0x043a29;
            when  4 result = 0x05aac3;
            when  5 result = 0x071f62;
            when  6 result = 0x08980f;
            when  7 result = 0x0a14d5;
            when  8 result = 0x0b95c2;
            when  9 result = 0x0d1adf;
            when 10 result = 0x0ea43a;
            when 11 result = 0x1031dc;
            when 12 result = 0x11c3d3;
            when 13 result = 0x135a2b;
            when 14 result = 0x14f4f0;
            when 15 result = 0x16942d;
            when 16 result = 0x1837f0;
            when 17 result = 0x19e046;
            when 18 result = 0x1b8d3a;
            when 19 result = 0x1d3eda;
            when 20 result = 0x1ef532;
            when 21 result = 0x20b051;
            when 22 result = 0x227043;
            when 23 result = 0x243516;
            when 24 result = 0x25fed7;
            when 25 result = 0x27cd94;
```

```
        when 26 result = 0x29a15b;
        when 27 result = 0x2b7a3a;
        when 28 result = 0x2d583f;
        when 29 result = 0x2f3b79;
        when 30 result = 0x3123f6;
        when 31 result = 0x3311c4;
        when 32 result = 0x3504f3;
        when 33 result = 0x36fd92;
        when 34 result = 0x38fbaf;
        when 35 result = 0x3aff5b;
        when 36 result = 0x3d08a4;
        when 37 result = 0x3f179a;
        when 38 result = 0x412c4d;
        when 39 result = 0x4346cd;
        when 40 result = 0x45672a;
        when 41 result = 0x478d75;
        when 42 result = 0x49b9be;
        when 43 result = 0x4bec15;
        when 44 result = 0x4e248c;
        when 45 result = 0x506334;
        when 46 result = 0x52a81e;
        when 47 result = 0x54f35b;
        when 48 result = 0x5744fd;
        when 49 result = 0x599d16;
        when 50 result = 0x5bfbb8;
        when 51 result = 0x5e60f5;
        when 52 result = 0x60ccdf;
        when 53 result = 0x633f89;
        when 54 result = 0x65b907;
        when 55 result = 0x68396a;
        when 56 result = 0x6ac0c7;
        when 57 result = 0x6d4f30;
        when 58 result = 0x6fe4ba;
        when 59 result = 0x728177;
        when 60 result = 0x75257d;
        when 61 result = 0x77d0df;
        when 62 result = 0x7a83b3;
        when 63 result = 0x7d3e0c;

else // N == 64
    case index of
        when  0 result = 0x0000000000000;
        when  1 result = 0x02C9A3E778061;
        when  2 result = 0x059B0D3158574;
        when  3 result = 0x0874518759BC8;
        when  4 result = 0x0B5586CF9890F;
        when  5 result = 0x0E3EC32D3D1A2;
        when  6 result = 0x11301D0125B51;
        when  7 result = 0x1429AAEA92DE0;
        when  8 result = 0x172B83C7D517B;
        when  9 result = 0x1A35BEB6FCB75;
        when 10 result = 0x1D4873168B9AA;
        when 11 result = 0x2063B88628CD6;
        when 12 result = 0x2387A6E756238;
        when 13 result = 0x26B4565E27CDD;
        when 14 result = 0x29E9DF51FDEE1;
        when 15 result = 0x2D285A6E4030B;
        when 16 result = 0x306FE0A31B715;
        when 17 result = 0x33C08B26416FF;
        when 18 result = 0x371A7373AA9CB;
        when 19 result = 0x3A7DB34E59FF7;
        when 20 result = 0x3DEA64C123422;
        when 21 result = 0x4160A21F72E2A;
        when 22 result = 0x44E086061892D;
        when 23 result = 0x486A2B5C13CD0;
        when 24 result = 0x4BFDAD5362A27;
        when 25 result = 0x4F9B2769D2CA7;
        when 26 result = 0x5342B569D4F82;
        when 27 result = 0x56F4736B527DA;
        when 28 result = 0x5AB07DD485429;
```

```
            when 29 result = 0x5E76F15AD2148;
            when 30 result = 0x6247EB03A5585;
            when 31 result = 0x6623882552225;
            when 32 result = 0x6A09E667F3BCD;
            when 33 result = 0x6DFB23C651A2F;
            when 34 result = 0x71F75E8EC5F74;
            when 35 result = 0x75FEB564267C9;
            when 36 result = 0x7A11473EB0187;
            when 37 result = 0x7E2F336CF4E62;
            when 38 result = 0x82589994CCE13;
            when 39 result = 0x868D99B4492ED;
            when 40 result = 0x8ACE5422AA0DB;
            when 41 result = 0x8F1AE99157736;
            when 42 result = 0x93737B0CDC5E5;
            when 43 result = 0x97D829FDE4E50;
            when 44 result = 0x9C49182A3F090;
            when 45 result = 0xA0C667B5DE565;
            when 46 result = 0xA5503B23E255D;
            when 47 result = 0xA9E6B5579FDBF;
            when 48 result = 0xAE89F995AD3AD;
            when 49 result = 0xB33A2B84F15FB;
            when 50 result = 0xB7F76F2FB5E47;
            when 51 result = 0xBCC1E904BC1D2;
            when 52 result = 0xC199BDD85529C;
            when 53 result = 0xC67F12E57D14B;
            when 54 result = 0xCB720DCEF9069;
            when 55 result = 0xD072D4A07897C;
            when 56 result = 0xD5818DCFBA487;
            when 57 result = 0xDA9E603DB3285;
            when 58 result = 0xDFC97337B9B5F;
            when 59 result = 0xE502EE78B3FF6;
            when 60 result = 0xEA4AFA2A490DA;
            when 61 result = 0xEFA1BEE615A27;
            when 62 result = 0xF50765B6E4540;
            when 63 result = 0xFA7C1819E90D8;

    return result<N-1:0>;
```

Library pseudocode for aarch64/functions/sve/FPLogB

```
// FPLogB()
// ========

bits(N) FPLogB(bits(N) op, FPCRType fpcr)
    assert N IN {16,32,64};
    integer result;
    (fptype,sign,value) = FPUnpack(op, fpcr);

    if fptype == FPType_SNaN || fptype == FPType_QNaN || fptype == FPType_Zero then
        FPProcessException(FPExc_InvalidOp, fpcr);
        result = -(2^(N-1));            // MinInt, 100..00
    elsif fptype == FPType_Infinity then
        result = 2^(N-1) - 1;           // MaxInt, 011..11
    else
        // FPUnpack has already scaled a subnormal input
        value = Abs(value);
        result = 0;
        while value < 1.0 do
            value = value * 2.0;
            result = result - 1;
        while value >= 2.0 do
            value = value / 2.0;
            result = result + 1;

        FPProcessDenorm(fptype, N, fpcr);

    return result<N-1:0>;
```

Library pseudocode for aarch64/functions/sve/FPMinNormal

```
// FPMinNormal()
// =============

bits(N) FPMinNormal(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = Zeros(E-1):'1';
    frac = Zeros(F);
    return sign : exp : frac;
```

Library pseudocode for aarch64/functions/sve/FPOne

```
// FPOne()
// =======

bits(N) FPOne(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '0':Ones(E-1);
    frac = Zeros(F);
    return sign : exp : frac;
```

Library pseudocode for aarch64/functions/sve/FPPointFive

```
// FPPointFive()
// =============

bits(N) FPPointFive(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '0':Ones(E-2):'0';
    frac = Zeros(F);
    return sign : exp : frac;
```

Library pseudocode for aarch64/functions/sve/FPProcess

```
// FPProcess()
// ===========

bits(N) FPProcess(bits(N) input)
    bits(N) result;
    assert N IN {16,32,64};
    FPCRType fpcr = FPCR[];
    (fptype,sign,value) = FPUnpack(input, fpcr);

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, input, fpcr);
    elsif fptype == FPType_Infinity then
        result = FPInfinity(sign, N);
    elsif fptype == FPType_Zero then
        result = FPZero(sign, N);
    else
        result = FPRound(value, fpcr, N);

        FPProcessDenorm(fptype, N, fpcr);

    return result;
```

Library pseudocode for aarch64/functions/sve/FPScale

```
// FPScale()
// =========

bits(N) FPScale(bits (N) op, integer scale, FPCRType fpcr)
    assert N IN {16,32,64};
    bits(N) result;
    (fptype,sign,value) = FPUnpack(op, fpcr);

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elsif fptype == FPType_Zero then
        result = FPZero(sign, N);
    elsif fptype == FPType_Infinity then
        result = FPInfinity(sign, N);
    else
        result = FPRound(value * (2.0^scale), fpcr, N);
        FPProcessDenorm(fptype, N, fpcr);

    return result;
```

Library pseudocode for aarch64/functions/sve/FPTrigMAdd

```
// FPTrigMAdd()
// ============

bits(N) FPTrigMAdd(integer x_in, bits(N) op1, bits(N) op2_in, FPCRType fpcr)
    assert N IN {16,32,64};
    bits(N) coeff;
    bits(N) op2 = op2_in;
    integer x = x_in;
    assert x >= 0;
    assert x < 8;

    if op2<N-1> == '1' then
        x = x + 8;

    coeff    = FPTrigMAddCoefficient[x, N];
    op2      = FPAbs(op2);
    result   = FPMulAdd(coeff, op1, op2, fpcr);
    return result;
```

Library pseudocode for aarch64/functions/sve/FPTrigMAddCoefficient

```
// FPTrigMAddCoefficient()
// =======================

bits(N) FPTrigMAddCoefficient[integer index, integer N]
    assert N IN {16,32,64};
    integer result;

    if N == 16 then
        case index of
            when  0 result = 0x3c00;
            when  1 result = 0xb155;
            when  2 result = 0x2030;
            when  3 result = 0x0000;
            when  4 result = 0x0000;
            when  5 result = 0x0000;
            when  6 result = 0x0000;
            when  7 result = 0x0000;
            when  8 result = 0x3c00;
            when  9 result = 0xb800;
            when 10 result = 0x293a;
            when 11 result = 0x0000;
            when 12 result = 0x0000;
            when 13 result = 0x0000;
            when 14 result = 0x0000;
            when 15 result = 0x0000;
    elsif N == 32 then
        case index of
            when  0 result = 0x3f800000;
            when  1 result = 0xbe2aaaab;
            when  2 result = 0x3c088886;
            when  3 result = 0xb95008b9;
            when  4 result = 0x36369d6d;
            when  5 result = 0x00000000;
            when  6 result = 0x00000000;
            when  7 result = 0x00000000;
            when  8 result = 0x3f800000;
            when  9 result = 0xbf000000;
            when 10 result = 0x3d2aaaa6;
            when 11 result = 0xbab60705;
            when 12 result = 0x37cd37cc;
            when 13 result = 0x00000000;
            when 14 result = 0x00000000;
            when 15 result = 0x00000000;
    else // N == 64
        case index of
            when  0 result = 0x3ff0000000000000;
            when  1 result = 0xbfc5555555555543;
            when  2 result = 0x3f8111111110f30c;
            when  3 result = 0xbf2a01a019b92fc6;
            when  4 result = 0x3ec71de351f3d22b;
            when  5 result = 0xbe5ae5e2b60f7b91;
            when  6 result = 0x3de5d8408868552f;
            when  7 result = 0x0000000000000000;
            when  8 result = 0x3ff0000000000000;
            when  9 result = 0xbfe0000000000000;
            when 10 result = 0x3fa5555555555536;
            when 11 result = 0xbf56c16c16c13a0b;
            when 12 result = 0x3efa01a019b1e8d8;
            when 13 result = 0xbe927e4f7282f468;
            when 14 result = 0x3e21ee96d2641b13;
            when 15 result = 0xbda8f76380fbb401;

    return result<N-1:0>;
```

Library pseudocode for aarch64/functions/sve/FPTrigSMul

```
// FPTrigSMul()
// ============

bits(N) FPTrigSMul(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    result = FPMul(op1, op1, fpcr);
    fpexc = FALSE;
    (fptype, sign, value) = FPUnpack(result, fpcr, fpexc);

    if !(fptype IN {FPType_QNaN, FPType_SNaN}) then
        result<N-1> = op2<0>;

    return result;
```

Library pseudocode for aarch64/functions/sve/FPTrigSSel

```
// FPTrigSSel()
// ============

bits(N) FPTrigSSel(bits(N) op1, bits(N) op2)
    assert N IN {16,32,64};
    bits(N) result;

    if op2<0> == '1' then
        result = FPOne(op2<1>, N);
    elsif op2<1> == '1' then
        result = FPNeg(op1);
    else
        result = op1;

    return result;
```

Library pseudocode for aarch64/functions/sve/FirstActive

```
// FirstActive()
// =============

bit FirstActive(bits(N) mask, bits(N) x, integer esize)
    integer elements = N DIV (esize DIV 8);
    for e = 0 to elements-1
        if ElemP[mask, e, esize] == '1' then return ElemP[x, e, esize];
    return '0';
```

Library pseudocode for aarch64/functions/sve/FloorPow2

```
// FloorPow2()
// ===========
// For a positive integer X, return the largest power of 2 <= X

integer FloorPow2(integer x)
    assert x >= 0;
    integer n = 1;
    if x == 0 then return 0;
    while x >= 2^n do
        n = n + 1;
    return 2^(n - 1);
```

Library pseudocode for aarch64/functions/sve/HaveSMEFullA64

```
// HaveSMEFullA64()
// ================
// Returns TRUE if the SME FA64 extension is implemented, FALSE otherwise.

boolean HaveSMEFullA64()
    return HaveSME() && boolean IMPLEMENTATION_DEFINED "Have SME FA64 extension";
```

Library pseudocode for aarch64/functions/sve/HaveSVE

```
// HaveSVE()
// =========

boolean HaveSVE()
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Have SVE ISA";
```

Library pseudocode for aarch64/functions/sve/HaveSVE2

```
// HaveSVE2()
// ==========
// Returns TRUE if the SVE2 extension is implemented, FALSE otherwise.

boolean HaveSVE2()
    return HaveSVE() && boolean IMPLEMENTATION_DEFINED "Have SVE2 extension";
```

Library pseudocode for aarch64/functions/sve/HaveSVE2AES

```
// HaveSVE2AES()
// =============
// Returns TRUE if the SVE2 AES extension is implemented, FALSE otherwise.

boolean HaveSVE2AES()
    return HaveSVE2() && boolean IMPLEMENTATION_DEFINED "Have SVE2 AES extension";
```

Library pseudocode for aarch64/functions/sve/HaveSVE2BitPerm

```
// HaveSVE2BitPerm()
// =================
// Returns TRUE if the SVE2 Bit Permissions extension is implemented, FALSE otherwise.

boolean HaveSVE2BitPerm()
    return HaveSVE2() && boolean IMPLEMENTATION_DEFINED "Have SVE2 BitPerm extension";
```

Library pseudocode for aarch64/functions/sve/HaveSVE2PMULL128

```
// HaveSVE2PMULL128()
// ==================
// Returns TRUE if the SVE2 128 bit PMULL extension is implemented, FALSE otherwise.

boolean HaveSVE2PMULL128()
    return HaveSVE2() && boolean IMPLEMENTATION_DEFINED "Have SVE2 128 bit PMULL extension";
```

Library pseudocode for aarch64/functions/sve/HaveSVE2SHA3

```
// HaveSVE2SHA3()
// ==============
// Returns TRUE if the SVE2 SHA3 extension is implemented, FALSE otherwise.

boolean HaveSVE2SHA3()
    return HaveSVE2() && boolean IMPLEMENTATION_DEFINED "Have SVE2 SHA3 extension";
```

Library pseudocode for aarch64/functions/sve/HaveSVE2SM4

```
// HaveSVE2SM4()
// =============
// Returns TRUE if the SVE2 SM4 extension is implemented, FALSE otherwise.

boolean HaveSVE2SM4()
    return HaveSVE2() && boolean IMPLEMENTATION_DEFINED "Have SVE2 SM4 extension";
```

Library pseudocode for aarch64/functions/sve/HaveSVEFP32MatMulExt

```
// HaveSVEFP32MatMulExt()
// ======================
// Returns TRUE if single-precision floating-point matrix multiply instruction support implemented and FA

boolean HaveSVEFP32MatMulExt()
    return HaveSVE() && boolean IMPLEMENTATION_DEFINED "Have SVE FP32 Matrix Multiply extension";
```

Library pseudocode for aarch64/functions/sve/HaveSVEFP64MatMulExt

```
// HaveSVEFP64MatMulExt()
// ======================
// Returns TRUE if double-precision floating-point matrix multiply instruction support implemented and FA

boolean HaveSVEFP64MatMulExt()
    return HaveSVE() && boolean IMPLEMENTATION_DEFINED "Have SVE FP64 Matrix Multiply extension";
```

Library pseudocode for aarch64/functions/sve/ImplementedSMEVectorLength

```
// ImplementedSMEVectorLength()
// ============================
// Reduce SVE/SME vector length to a supported value (power of two)

integer ImplementedSMEVectorLength(integer nbits_in)
    integer maxbits = MaxImplementedSVL();
    assert 128 <= maxbits && maxbits <= 2048 && IsPow2(maxbits);
    integer nbits = Min(nbits_in, maxbits);
    assert 128 <= nbits && nbits <= 2048 && Align(nbits, 128) == nbits;

    // Search for a supported power-of-two VL less than or equal to nbits
    while nbits > 128 do
        if IsPow2(nbits) && SupportedPowerTwoSVL(nbits) then return nbits;
        nbits = nbits - 128;

    // Return the smallest supported power-of-two VL
    nbits = 128;
    while nbits < maxbits do
        if SupportedPowerTwoSVL(nbits) then return nbits;
        nbits = nbits * 2;

    // The only option is maxbits
    return maxbits;
```

Library pseudocode for aarch64/functions/sve/ImplementedSVEVectorLength

```
// ImplementedSVEVectorLength()
// ============================
// Reduce SVE vector length to a supported value (e.g. power of two)

integer ImplementedSVEVectorLength(integer nbits_in)
    integer nbits = Min(nbits_in, MaxImplementedVL());
    assert 128 <= nbits && nbits <= 2048 && Align(nbits, 128) == nbits;

    while nbits > 128 do
        if IsPow2(nbits) || SupportedNonPowerTwoVL(nbits) then return nbits;
        nbits = nbits - 128;
    return nbits;
```

Library pseudocode for aarch64/functions/sve/InStreamingMode

```
// InStreamingMode()
// =================

boolean InStreamingMode()
    return HaveSME() && PSTATE.SM == '1';
```

Library pseudocode for aarch64/functions/sve/IsEven

```
// IsEven()
// ========

boolean IsEven(integer val)
    return val MOD 2 == 0;
```

Library pseudocode for aarch64/functions/sve/IsFPEnabled

```
// IsFPEnabled()
// =============
// Returns TRUE if accesses to the Advanced SIMD and floating-point
// registers are enabled at the target exception level in the current
// execution state and FALSE otherwise.

boolean IsFPEnabled(bits(2) el)
    if ELUsingAArch32(el) then
        return AArch32.IsFPEnabled(el);
    else
        return AArch64.IsFPEnabled(el);
```

Library pseudocode for aarch64/functions/sve/IsFullA64Enabled

```
// IsFullA64Enabled()
// ==================
// Returns TRUE is full A64 is enabled in Streaming mode and FALSE othersise.

boolean IsFullA64Enabled()
    if !HaveSMEFullA64() then return FALSE;

    // Check if full SVE disabled in SMCR_EL1
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check full SVE at EL0/EL1
        if SMCR_EL1.FA64 == '0' then return FALSE;

    // Check if full SVE disabled in SMCR_EL2
    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        if SMCR_EL2.FA64 == '0' then return FALSE;

    // Check if full SVE disabled in SMCR_EL3
    if HaveEL(EL3) then
        if SMCR_EL3.FA64 == '0' then return FALSE;

    return TRUE;
```

Library pseudocode for aarch64/functions/sve/IsNormalSVEEnabled

```
// IsNormalSVEEnabled()
// ====================
// Returns TRUE if access to normal SVE is enabled at the target
// exception level and FALSE otherwise.

boolean IsNormalSVEEnabled(bits(2) el)
    boolean disabled;
    if ELUsingAArch32(el) then
        return FALSE;

    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} && !IsInHost() then
        // Check SVE at EL0/EL1
        case CPACR_EL1.ZEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    // Check if access disabled in CPTR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            case CPTR_EL2.ZEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = el == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then return FALSE;
        else
            if CPTR_EL2.TZ == '1' then return FALSE;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.EZ == '0' then return FALSE;

    return TRUE;
```

Library pseudocode for aarch64/functions/sve/IsOdd

```
// IsOdd()
// =======

boolean IsOdd(integer val)
    return val MOD 2 == 1;
```

Library pseudocode for aarch64/functions/sve/IsPow2

```
// IsPow2()
// ========
// Return TRUE if positive integer X is a power of 2. Otherwise,
// return FALSE.

boolean IsPow2(integer x)
    if x <= 0 then return FALSE;
    return FloorPow2(x) == CeilPow2(x);
```

Library pseudocode for aarch64/functions/sve/IsSVEEnabled

```
// IsSVEEnabled()
// ==============
// Returns TRUE if access to SVE instructions and System registers is
// enabled at the target exception level and FALSE otherwise.

boolean IsSVEEnabled(bits(2) el)
    if HaveSME() && PSTATE.SM == '1' then
        return IsStreamingSVEEnabled(el);
    elsif HaveSVE() then
        return IsNormalSVEEnabled(el);
    else
        return FALSE;
```

Library pseudocode for aarch64/functions/sve/IsStreamingSVEEnabled

```
// IsStreamingSVEEnabled()
// =======================
// Returns TRUE if access to streaming SVE is enabled at the
// target exception level and FALSE otherwise.

boolean IsStreamingSVEEnabled(bits(2) el)
    boolean disabled;
    if ELUsingAArch32(el) then
        return FALSE;

    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} && !IsInHost() then
        // Check SME at EL0/EL1
        case CPACR_EL1.SMEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    // Check if access disabled in CPTR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            case CPTR_EL2.SMEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = el == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then return FALSE;
        else
            if CPTR_EL2.TSM == '1' then return FALSE;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.ESM == '0' then return FALSE;

    return TRUE;
```

Library pseudocode for aarch64/functions/sve/LastActive

```
// LastActive()
// ============

bit LastActive(bits(N) mask, bits(N) x, integer esize)
    integer elements = N DIV (esize DIV 8);
    for e = elements-1 downto 0
        if ElemP[mask, e, esize] == '1' then return ElemP[x, e, esize];
    return '0';
```

Library pseudocode for aarch64/functions/sve/LastActiveElement

```
// LastActiveElement()
// ===================

integer LastActiveElement(bits(N) mask, integer esize)
    integer elements = N DIV (esize DIV 8);
    for e = elements-1 downto 0
        if ElemP[mask, e, esize] == '1' then return e;
    return -1;
```

Library pseudocode for aarch64/functions/sve/MaxImplementedSVL

```
// MaxImplementedSVL()
// ===================

integer MaxImplementedSVL()
    return integer IMPLEMENTATION_DEFINED;
```

Library pseudocode for aarch64/functions/sve/MaxImplementedVL

```
// MaxImplementedVL()
// ==================

integer MaxImplementedVL()
    return integer IMPLEMENTATION_DEFINED;
```

Library pseudocode for aarch64/functions/sve/MaybeZeroSVEUppers

```
// MaybeZeroSVEUppers()
// ====================

MaybeZeroSVEUppers(bits(2) target_el)
    boolean lower_enabled;

    if UInt(target_el) <= UInt(PSTATE.EL) || !IsSVEEnabled(target_el) then
        return;

    if target_el == EL3 then
        if EL2Enabled() then
            lower_enabled = IsFPEnabled(EL2);
        else
            lower_enabled = IsFPEnabled(EL1);
    elsif target_el == EL2 then
        assert !ELUsingAArch32(EL2);
        if HCR_EL2.TGE == '0' then
            lower_enabled = IsFPEnabled(EL1);
        else
            lower_enabled = IsFPEnabled(EL0);
    else
        assert target_el == EL1 && !ELUsingAArch32(EL1);
        lower_enabled = IsFPEnabled(EL0);

    if lower_enabled then
        integer vl = if IsSVEEnabled(PSTATE.EL) then CurrentVL else 128;
        integer pl = vl DIV 8;
        for n = 0 to 31
            if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
                _Z[n] = ZeroExtend(_Z[n]<vl-1:0>, MAX_VL);
        for n = 0 to 15
            if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
                _P[n] = ZeroExtend(_P[n]<pl-1:0>, MAX_PL);
        if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
            _FFR = ZeroExtend(_FFR<pl-1:0>, MAX_PL);
        if HaveSME() && PSTATE.ZA == '1' then
            integer accessiblevecs = SVL DIV 8;
            integer allvecs = MaxImplementedSVL() DIV 8;

            for n = 0 to accessiblevecs - 1
                if ConstrainUnpredictableBool(Unpredictable_SMEZEROUPPER) then
                    _ZA[n] = ZeroExtend(_ZA[n]<SVL-1:0>, MAX_VL);
            for n = accessiblevecs to allvecs - 1
                if ConstrainUnpredictableBool(Unpredictable_SMEZEROUPPER) then
                    _ZA[n] = Zeros(MAX_VL);
```

Library pseudocode for aarch64/functions/sve/MemNF

```
// MemNF[] - non-assignment form
// ============================

(bits(8*size), boolean) MemNF[bits(64) address, integer size, AccType acctype]
    assert size IN {1, 2, 4, 8, 16};
    bits(8*size) value;
    boolean bad;

    aligned = (address == Align(address, size));
    A = SCTLR[].A;

    if !aligned && (A == '1') then
        return (bits(8*size) UNKNOWN, TRUE);

    atomic = aligned || size == 1;

    if !atomic then
        (value<7:0>, bad) = MemSingleNF[address, 1, acctype, aligned];

        if bad then
            return (bits(8*size) UNKNOWN, TRUE);

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        if !aligned then
            c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
            assert c IN {Constraint_FAULT, Constraint_NONE};
            if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            (value<8*i+7:8*i>, bad) = MemSingleNF[address+i, 1, acctype, aligned];

            if bad then
                return (bits(8*size) UNKNOWN, TRUE);
    else
        (value, bad) = MemSingleNF[address, size, acctype, aligned];
        if bad then
            return (bits(8*size) UNKNOWN, TRUE);

    if BigEndian(acctype) then
        value = BigEndianReverse(value);

    return (value, FALSE);
```

Library pseudocode for aarch64/functions/sve/MemSingleNF

```
// MemSingleNF[] - non-assignment form
// ===================================

(bits(8*size), boolean) MemSingleNF[bits(64) address, integer size, AccType acctype, boolean aligned]
    assert acctype IN {AccType_CNOTFIRST, AccType_NONFAULT};
    bits(8*size) value;
    boolean iswrite = FALSE;
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus memstatus;
    FaultRecord fault;

    // Implementation may suppress NF load for any reason
    if ConstrainUnpredictableBool(Unpredictable_NONFAULT) then
        return (bits(8*size) UNKNOWN, TRUE);

    boolean istagaccess = HaveMTE2Ext() && AArch64.AccessIsTagChecked(address, acctype);

    // MMU or MPU
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Non-fault load from Device memory must not be performed externally
    if memaddrdesc.memattrs.memtype == MemType_Device then
        return (bits(8*size) UNKNOWN, TRUE);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return (bits(8*size) UNKNOWN, TRUE);

    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);
    if HaveTME() then
        accdesc.transactional = TSTATE.depth > 0;
    boolean istagchecked = istagaccess;
    if istagchecked then
        bits(4) ptag = AArch64.PhysicalTag(address);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
            return (bits(8*size) UNKNOWN, TRUE);

    (memstatus, value) = PhysMemRead(memaddrdesc, size, accdesc);
    if IsFault(memstatus) then
        if IsExternalAbortTakenSynchronously(memstatus, iswrite, memaddrdesc,
                                             size, accdesc) then
            return (bits(8*size) UNKNOWN, TRUE);
        fault = NoFault();
        fault.errortype = memstatus.errortype;
        fault.acctype = memstatus.acctype;
        fault.extflag = memstatus.extflag;
        fault.statuscode = memstatus.statuscode;
        PendSErrorInterrupt(fault);

    return (value, FALSE);
```

Library pseudocode for aarch64/functions/sve/NVL

```
// NVL - non-assignment form
// ========================
// Normal VL

integer NVL
    integer vl;

    if PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost()) then
        vl = UInt(ZCR_EL1.LEN);

    if PSTATE.EL == EL2 || (PSTATE.EL == EL0 && IsInHost()) then
        vl = UInt(ZCR_EL2.LEN);
    elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        vl = Min(vl, UInt(ZCR_EL2.LEN));

    if PSTATE.EL == EL3 then
        vl = UInt(ZCR_EL3.LEN);
    elsif HaveEL(EL3) then
        vl = Min(vl, UInt(ZCR_EL3.LEN));

    vl = (vl + 1) * 128;
    vl = ImplementedSVEVectorLength(vl);

    return vl;
```

Library pseudocode for aarch64/functions/sve/NoneActive

```
// NoneActive()
// ============

bit NoneActive(bits(N) mask, bits(N) x, integer esize)
    integer elements = N DIV (esize DIV 8);
    for e = 0 to elements-1
        if ElemP[mask, e, esize] == '1' && ElemP[x, e, esize] == '1' then return '0';
    return '1';
```

Library pseudocode for aarch64/functions/sve/P

```
// P[] - non-assignment form
// =========================

bits(width) P[integer n, integer width]
    assert n >= 0 && n <= 31;
    assert width == CurrentVL DIV 8;
    return _P[n]<width-1:0>;

// P[] - assignment form
// =====================

P[integer n, integer width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width == CurrentVL DIV 8;
    if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
        _P[n] = ZeroExtend(value, MAX_PL);
    else
        _P[n]<width-1:0> = value;
```

Library pseudocode for aarch64/functions/sve/PredTest

```
// PredTest()
// ==========

bits(4) PredTest(bits(N) mask, bits(N) result, integer esize)
    bit n = FirstActive(mask, result, esize);
    bit z = NoneActive(mask, result, esize);
    bit c = NOT LastActive(mask, result, esize);
    bit v = '0';
    return n:z:c:v;
```

Library pseudocode for aarch64/functions/sve/ReducePredicated

```
// ReducePredicated()
// ==================

bits(esize) ReducePredicated(ReduceOp op, bits(N) input, bits(M) mask, bits(esize) identity)
    assert(N == M * 8);
    integer p2bits = CeilPow2(N);
    bits(p2bits) operand;
    integer elements = p2bits DIV esize;

    for e = 0 to elements-1
        if e * esize < N && ElemP[mask, e, esize] == '1' then
            Elem[operand, e, esize] = Elem[input, e, esize];
        else
            Elem[operand, e, esize] = identity;

    return Reduce(op, operand, esize);
```

Library pseudocode for aarch64/functions/sve/ResetSMEState

```
// ResetSMEState()
// ===============

ResetSMEState()
    integer vectors = MAX_VL DIV 8;
    for n = 0 to vectors - 1
        _ZA[n] = Zeros(MAX_VL);
```

Library pseudocode for aarch64/functions/sve/ResetSVEState

```
// ResetSVEState()
// ===============

ResetSVEState()
    for n = 0 to 31
        _Z[n] = Zeros(MAX_VL);
    for n = 0 to 15
        _P[n] = Zeros(MAX_PL);
    _FFR = Zeros(MAX_PL);
    FPSR = ZeroExtend(0x0800009f<31:0>, 64);
```

Library pseudocode for aarch64/functions/sve/Reverse

```
// Reverse()
// =========
// Reverse subwords of M bits in an N-bit word

bits(N) Reverse(bits(N) word, integer M)
    bits(N) result;
    integer sw = N DIV M;
    assert N == sw * M;
    for s = 0 to sw-1
        Elem[result, (sw - 1) - s, M] = Elem[word, s, M];
    return result;
```

Library pseudocode for aarch64/functions/sve/SMEAccessTrap

```
// SMEAccessTrap()
// ===============
// Trapped access to SME registers due to CPACR_EL1, CPTR_EL2, or CPTR_EL3.

SMEAccessTrap(SMEExceptionType etype, bits(2) target_el_in)
    bits(2) target_el = target_el_in;
    assert UInt(target_el) >= UInt(PSTATE.EL);
    if target_el == EL0 then
        target_el = EL1;
    boolean route_to_el2 = PSTATE.EL == EL0 && target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1';

    exception = ExceptionSyndrome(Exception_SMEAccessTrap);
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    case etype of
        when SMEExceptionType_AccessTrap
            exception.syndrome<1:0> = '00';
        when SMEExceptionType_Streaming
            exception.syndrome<1:0> = '01';
        when SMEExceptionType_NotStreaming
            exception.syndrome<1:0> = '10';
        when SMEExceptionType_InactiveZA
            exception.syndrome<1:0> = '11';

    if route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/functions/sve/SMEExceptionType

```
enumeration SMEExceptionType {
    SMEExceptionType_AccessTrap,        // SME functionality trapped or disabled
    SMEExceptionType_Streaming,         // Illegal instruction in Streaming SVE mode
    SMEExceptionType_NotStreaming,      // Illegal instruction not in Streaming SVE mode
    SMEExceptionType_InactiveZA,        // Illegal instruction when ZA is inactive
};
```

Library pseudocode for aarch64/functions/sve/SVEAccessTrap

```
// SVEAccessTrap()
// ===============
// Trapped access to SVE registers due to CPACR_EL1, CPTR_EL2, or CPTR_EL3.

SVEAccessTrap(bits(2) target_el)
    assert UInt(target_el) >= UInt(PSTATE.EL) && target_el != EL0 && HaveEL(target_el);
    route_to_el2 = target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1';

    exception = ExceptionSyndrome(Exception_SVEAccessTrap);
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    if route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/functions/sve/SVECmp

```
enumeration SVECmp { Cmp_EQ, Cmp_NE, Cmp_GE, Cmp_GT, Cmp_LT, Cmp_LE, Cmp_UN };
```

Library pseudocode for aarch64/functions/sve/SVEMoveMaskPreferred

```
// SVEMoveMaskPreferred()
// ======================
// Return FALSE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single DUP instruction.
// Used as a condition for the preferred MOV<-DUPM alias.

boolean SVEMoveMaskPreferred(bits(13) imm13)
    bits(64) imm;
    (imm, -) = DecodeBitMasks(imm13<12>, imm13<5:0>, imm13<11:6>, TRUE, 64);

    // Check for 8 bit immediates
    if !IsZero(imm<7:0>) then
        // Check for 'ffffffffffffffxy' or '00000000000000xy'
        if IsZero(imm<63:7>) || IsOnes(imm<63:7>) then
            return FALSE;

        // Check for 'ffffffxyffffffxy' or '000000xy000000xy'
        if imm<63:32> == imm<31:0> && (IsZero(imm<31:7>) || IsOnes(imm<31:7>)) then
            return FALSE;

        // Check for 'ffxyffxyffxyffxy' or '00xy00xy00xy00xy'
        if imm<63:32> == imm<31:0> && imm<31:16> == imm<15:0> && (IsZero(imm<15:7>) || IsOnes(imm<15:7>))
            return FALSE;

        // Check for 'xyxyxyxyxyxyxyxy'
        if imm<63:32> == imm<31:0> && imm<31:16> == imm<15:0> && (imm<15:8> == imm<7:0>) then
            return FALSE;

    // Check for 16 bit immediates
    else
        // Check for 'ffffffffffffxy00' or '000000000000xy00'
        if IsZero(imm<63:15>) || IsOnes(imm<63:15>) then
            return FALSE;

        // Check for 'ffffxy00ffffxy00' or '0000xy000000xy00'
        if imm<63:32> == imm<31:0> && (IsZero(imm<31:7>) || IsOnes(imm<31:7>)) then
            return FALSE;

        // Check for 'xy00xy00xy00xy00'
        if imm<63:32> == imm<31:0> && imm<31:16> == imm<15:0> then
            return FALSE;

    return TRUE;
```

Library pseudocode for aarch64/functions/sve/SVL

```
// SVL - non-assignment form
// ========================
// Streaming SVL

integer SVL
    integer vl;

    if PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost()) then
        vl = UInt(SMCR_EL1.LEN);

    if PSTATE.EL == EL2 || (PSTATE.EL == EL0 && IsInHost()) then
        vl = UInt(SMCR_EL2.LEN);
    elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        vl = Min(vl, UInt(SMCR_EL2.LEN));

    if PSTATE.EL == EL3 then
        vl = UInt(SMCR_EL3.LEN);
    elsif HaveEL(EL3) then
        vl = Min(vl, UInt(SMCR_EL3.LEN));

    vl = (vl + 1) * 128;
    vl = ImplementedSMEVectorLength(vl);

    return vl;
```

Library pseudocode for aarch64/functions/sve/SetPSTATE_SM

```
// SetPSTATE_SM()
// ==============

SetPSTATE_SM(bit value)
    if PSTATE.SM != value then
        ResetSVEState();
        PSTATE.SM = value;
```

Library pseudocode for aarch64/functions/sve/SetPSTATE_SVCR

```
// SetPSTATE_SVCR
// ==============

SetPSTATE_SVCR(bits(32) svcr)
    SetPSTATE_SM(svcr<0>);
    SetPSTATE_ZA(svcr<1>);
```

Library pseudocode for aarch64/functions/sve/SetPSTATE_ZA

```
// SetPSTATE_ZA()
// ==============

SetPSTATE_ZA(bit value)
    if PSTATE.ZA != value then
        ResetSMEState();
        PSTATE.ZA = value;
```

Library pseudocode for aarch64/functions/sve/ShiftSat

```
// ShiftSat()
// ==========

integer ShiftSat(integer shift, integer esize)
    if shift > esize+1 then return esize+1;
    elsif shift < -(esize+1) then return -(esize+1);
    return shift;
```

Library pseudocode for aarch64/functions/sve/SupportedNonPowerTwoVL

```
// SupportedNonPowerTwoVL()
// ========================

boolean SupportedNonPowerTwoVL(integer nbits)
    return boolean IMPLEMENTATION_DEFINED;
```

Library pseudocode for aarch64/functions/sve/SupportedPowerTwoSVL

```
// SupportedPowerTwoSVL()
// ======================

boolean SupportedPowerTwoSVL(integer nbits)
    return boolean IMPLEMENTATION_DEFINED;
```

Library pseudocode for aarch64/functions/sve/System

```
constant integer MAX_VL = 2048;
constant integer MAX_PL = 256;
array bits(MAX_VL) _Z[0..31];
array bits(MAX_PL) _P[0..15];
bits(MAX_PL) _FFR;
```

Library pseudocode for aarch64/functions/sve/Z

```
// Z[] - non-assignment form
// =========================

bits(width) Z[integer n, integer width]
    assert n >= 0 && n <= 31;
    assert width == CurrentVL;
    return _Z[n]<width-1:0>;

// Z[] - assignment form
// =====================

Z[integer n, integer width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width == CurrentVL;
    if ConstrainUnpredictableBool(Unpredictable_SVEZEROUPPER) then
        _Z[n] = ZeroExtend(value, MAX_VL);
    else
        _Z[n]<width-1:0> = value;
```

Library pseudocode for aarch64/functions/sysregisters/CNTKCTL

```
// CNTKCTL[] - non-assignment form
// ===============================

CNTKCTLType CNTKCTL[]
    bits(64) r;
    if IsInHost() then
        r = CNTHCTL_EL2;
        return r;
    r = CNTKCTL_EL1;
    return r;
```

Library pseudocode for aarch64/functions/sysregisters/CNTKCTLType

```
type CNTKCTLType;
```

Library pseudocode for aarch64/functions/sysregisters/CPACR

```
// CPACR[] - non-assignment form
// ============================

CPACRType CPACR[]
    bits(64) r;
    if IsInHost() then
        r = CPTR_EL2;
        return r;
    r = CPACR_EL1;
    return r;
```

Library pseudocode for aarch64/functions/sysregisters/CPACRType

```
type CPACRType;
```

Library pseudocode for aarch64/functions/sysregisters/ELR

```
// ELR[] - non-assignment form
// ===========================

bits(64) ELR[bits(2) el]
    bits(64) r;
    case el of
        when EL1  r = ELR_EL1;
        when EL2  r = ELR_EL2;
        when EL3  r = ELR_EL3;
        otherwise Unreachable();
    return r;

// ELR[] - non-assignment form
// ===========================

bits(64) ELR[]
    assert PSTATE.EL != EL0;
    return ELR[PSTATE.EL];

// ELR[] - assignment form
// =======================

ELR[bits(2) el] = bits(64) value
    bits(64) r = value;
    case el of
        when EL1  ELR_EL1 = r;
        when EL2  ELR_EL2 = r;
        when EL3  ELR_EL3 = r;
        otherwise Unreachable();
    return;

// ELR[] - assignment form
// =======================

ELR[] = bits(64) value
    assert PSTATE.EL != EL0;
    ELR[PSTATE.EL] = value;
    return;
```

Library pseudocode for aarch64/functions/sysregisters/ESR

```
// ESR[] - non-assignment form
// ==========================

ESRType ESR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = ESR_EL1;
        when EL2  r = ESR_EL2;
        when EL3  r = ESR_EL3;
        otherwise Unreachable();
    return r;

// ESR[] - non-assignment form
// ==========================

ESRType ESR[]
    return ESR[S1TranslationRegime()];

// ESR[] - assignment form
// =======================

ESR[bits(2) regime] = ESRType value
    bits(64) r = value;
    case regime of
        when EL1  ESR_EL1 = r;
        when EL2  ESR_EL2 = r;
        when EL3  ESR_EL3 = r;
        otherwise Unreachable();
    return;

// ESR[] - assignment form
// =======================

ESR[] = ESRType value
    ESR[S1TranslationRegime()] = value;
```

Library pseudocode for aarch64/functions/sysregisters/ESRType

```
type ESRType;
```

Library pseudocode for aarch64/functions/sysregisters/FAR

```
// FAR[] - non-assignment form
// ===========================

bits(64) FAR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = FAR_EL1;
        when EL2  r = FAR_EL2;
        when EL3  r = FAR_EL3;
        otherwise Unreachable();
    return r;

// FAR[] - non-assignment form
// ===========================

bits(64) FAR[]
    return FAR[S1TranslationRegime()];

// FAR[] - assignment form
// =======================

FAR[bits(2) regime] = bits(64) value
    bits(64) r = value;
    case regime of
        when EL1  FAR_EL1 = r;
        when EL2  FAR_EL2 = r;
        when EL3  FAR_EL3 = r;
        otherwise Unreachable();
    return;

// FAR[] - assignment form
// =======================

FAR[] = bits(64) value
    FAR[S1TranslationRegime()] = value;
    return;
```

Library pseudocode for aarch64/functions/sysregisters/MAIR

```
// MAIR[] - non-assignment form
// ============================

MAIRType MAIR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = MAIR_EL1;
        when EL2  r = MAIR_EL2;
        when EL3  r = MAIR_EL3;
        otherwise Unreachable();
    return r;

// MAIR[] - non-assignment form
// ============================

MAIRType MAIR[]
    return MAIR[S1TranslationRegime()];
```

Library pseudocode for aarch64/functions/sysregisters/MAIRType

```
type MAIRType;
```

Library pseudocode for aarch64/functions/sysregisters/SCTLR

```
// SCTLR[] - non-assignment form
// ============================

SCTLRType SCTLR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = SCTLR_EL1;
        when EL2  r = SCTLR_EL2;
        when EL3  r = SCTLR_EL3;
        otherwise Unreachable();
    return r;

// SCTLR[] - non-assignment form
// ============================

SCTLRType SCTLR[]
    return SCTLR[S1TranslationRegime()];
```

Library pseudocode for aarch64/functions/sysregisters/SCTLRType

```
type SCTLRType;
```

Library pseudocode for aarch64/functions/sysregisters/VBAR

```
// VBAR[] - non-assignment form
// ============================

bits(64) VBAR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = VBAR_EL1;
        when EL2  r = VBAR_EL2;
        when EL3  r = VBAR_EL3;
        otherwise Unreachable();
    return r;

// VBAR[] - non-assignment form
// ============================

bits(64) VBAR[]
    return VBAR[S1TranslationRegime()];
```

Library pseudocode for aarch64/functions/system/AArch64.AllocationTagAccessIsEnabled

```
// AArch64.AllocationTagAccessIsEnabled()
// ======================================
// Check whether access to Allocation Tags is enabled.

boolean AArch64.AllocationTagAccessIsEnabled(AccType acctype)
    bits(2) el = AArch64.AccessUsesEL(acctype);

    if SCR_EL3.ATA == '0' && el IN {EL0, EL1, EL2} then
        return FALSE;
    elsif HCR_EL2.ATA == '0' && el IN {EL0, EL1} && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' then
        return FALSE;
    elsif SCTLR_EL3.ATA == '0' && el == EL3 then
        return FALSE;
    elsif SCTLR_EL2.ATA == '0' && el == EL2 then
        return FALSE;
    elsif SCTLR_EL1.ATA == '0' && el == EL1 then
        return FALSE;
    elsif SCTLR_EL2.ATA0 == '0' && el == EL0 && EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' then
        return FALSE;
    elsif SCTLR_EL1.ATA0 == '0' && el == EL0 && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') then
        return FALSE;
    else
        return TRUE;
```

Library pseudocode for aarch64/functions/system/AArch64.CheckSystemAccess

```
// AArch64.CheckSystemAccess()
// ===========================

AArch64.CheckSystemAccess(bits(2) op0, bits(3) op1, bits(4) crn,
                          bits(4) crm, bits(3) op2, bits(5) rt, bit read)
    if (TSTATE.depth > 0 &&
          !CheckTransactionalSystemAccess(op0, op1, crn, crm, op2, read)) then
        FailTransaction(TMFailure_ERR, FALSE);

    return;
```

Library pseudocode for aarch64/functions/system/AArch64.ChooseNonExcludedTag

```
// AArch64.ChooseNonExcludedTag()
// ==============================
// Return a tag derived from the start and the offset values, excluding
// any tags in the given mask.

bits(4) AArch64.ChooseNonExcludedTag(bits(4) tag_in, bits(4) offset_in, bits(16) exclude)
    bits(4) tag = tag_in;
    bits(4) offset = offset_in;

    if IsOnes(exclude) then
        return '0000';

    if offset == '0000' then
        while exclude<UInt(tag)> == '1' do
            tag = tag + '0001';

    while offset != '0000' do
        offset = offset - '0001';
        tag = tag + '0001';
        while exclude<UInt(tag)> == '1' do
            tag = tag + '0001';

    return tag;
```

Library pseudocode for aarch64/functions/system/AArch64.ExecutingBROrBLROrRetInstr

```
// AArch64.ExecutingBROrBLROrRetInstr()
// ====================================
// Returns TRUE if current instruction is a BR, BLR, RET, B[L]RA[B][Z], or RETA[B].

boolean AArch64.ExecutingBROrBLROrRetInstr()
    if !HaveBTIExt() then return FALSE;

    instr = ThisInstr();
    if instr<31:25> == '1101011' && instr<20:16> == '11111' then
        opc = instr<24:21>;
        return opc != '0101';
    else
        return FALSE;
```

Library pseudocode for aarch64/functions/system/AArch64.ExecutingBTIInstr

```
// AArch64.ExecutingBTIInstr()
// ===========================
// Returns TRUE if current instruction is a BTI.

boolean AArch64.ExecutingBTIInstr()
    if !HaveBTIExt() then return FALSE;

    instr = ThisInstr();
    if instr<31:22> == '1101010100' && instr<21:12> == '0000110010' && instr<4:0> == '11111' then
        CRm  = instr<11:8>;
        op2  = instr<7:5>;
        return (CRm == '0100' && op2<0> == '0');
    else
        return FALSE;
```

Library pseudocode for aarch64/functions/system/AArch64.ExecutingERETInstr

```
// AArch64.ExecutingERETInstr()
// ============================
// Returns TRUE if current instruction is ERET.

boolean AArch64.ExecutingERETInstr()
    instr = ThisInstr();
    return instr<31:12> == '11010110100111110000';
```

Library pseudocode for aarch64/functions/system/AArch64.ImpDefSysInstr

```
// Execute an implementation-defined system instruction with write (source operand).
AArch64.ImpDefSysInstr(integer el, bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2, integer t);
```

Library pseudocode for aarch64/functions/system/AArch64.ImpDefSysInstrWithResult

```
// Execute an implementation-defined system instruction with read (result operand).
AArch64.ImpDefSysInstrWithResult(integer el, bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2);
```

Library pseudocode for aarch64/functions/system/AArch64.ImpDefSysRegRead

```
// Read from an implementation-defined System register and write the contents of the register to X[t].
AArch64.ImpDefSysRegRead(bits(2) op0, bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2, integer t);
```

Library pseudocode for aarch64/functions/system/AArch64.ImpDefSysRegWrite

```
// Write to an implementation-defined System register.
AArch64.ImpDefSysRegWrite(bits(2) op0, bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2, integer t);
```

Library pseudocode for aarch64/functions/system/AArch64.NextRandomTagBit

```
// AArch64.NextRandomTagBit()
// ==========================
// Generate a random bit suitable for generating a random Allocation Tag.

bit AArch64.NextRandomTagBit()
    bits(16) lfsr = RGSR_EL1.SEED;
    bit top = lfsr<5> EOR lfsr<3> EOR lfsr<2> EOR lfsr<0>;
    RGSR_EL1.SEED = top:lfsr<15:1>;
    return top;
```

Library pseudocode for aarch64/functions/system/AArch64.RandomTag

```
// AArch64.RandomTag()
// ===================
// Generate a random Allocation Tag.

bits(4) AArch64.RandomTag()
    bits(4) tag;
    for i = 0 to 3
        tag<i> = AArch64.NextRandomTagBit();
    return tag;
```

Library pseudocode for aarch64/functions/system/AArch64.SysInstr

```
// Execute a system instruction with write (source operand).
AArch64.SysInstr(integer op0, integer op1, integer crn, integer crm, integer op2, integer t);
```

Library pseudocode for aarch64/functions/system/AArch64.SysInstrWithResult

```
// Execute a system instruction with read (result operand).
// Writes the result of the instruction to X[t].
AArch64.SysInstrWithResult(integer op0, integer op1, integer crn, integer crm, integer op2, integer t);
```

Library pseudocode for aarch64/functions/system/AArch64.SysRegRead

```
// Read from a System register and write the contents of the register to X[t].
AArch64.SysRegRead(integer op0, integer op1, integer crn, integer crm, integer op2, integer t);
```

Library pseudocode for aarch64/functions/system/AArch64.SysRegWrite

```
// Write to a System register.
AArch64.SysRegWrite(integer op0, integer op1, integer crn, integer crm, integer op2, integer t);
```

Library pseudocode for aarch64/functions/system/BTypeCompatible

```
boolean BTypeCompatible;
```

Library pseudocode for aarch64/functions/system/BTypeCompatible_BTI

```
// BTypeCompatible_BTI
// ==================
// This function determines whether a given hint encoding is compatible with the current value of
// PSTATE.BTYPE. A value of TRUE here indicates a valid Branch Target Identification instruction.

boolean BTypeCompatible_BTI(bits(2) hintcode)
    case hintcode of
        when '00'
            return FALSE;
        when '01'
            return PSTATE.BTYPE != '11';
        when '10'
            return PSTATE.BTYPE != '10';
        when '11'
            return TRUE;
```

Library pseudocode for aarch64/functions/system/BTypeCompatible_PACIXSP

```
// BTypeCompatible_PACIXSP()
// =========================
// Returns TRUE if PACIASP, PACIBSP instruction is implicit compatible with PSTATE.BTYPE,
// FALSE otherwise.

boolean BTypeCompatible_PACIXSP()
    if PSTATE.BTYPE IN {'01', '10'} then
        return TRUE;
    elsif PSTATE.BTYPE == '11' then
        index = if PSTATE.EL == EL0 then 35 else 36;
        return SCTLR[]<index> == '0';
    else
        return FALSE;
```

Library pseudocode for aarch64/functions/system/BTypeNext

```
bits(2) BTypeNext;
```

Library pseudocode for aarch64/functions/system/ChooseRandomNonExcludedTag

```
// The ChooseRandomNonExcludedTag function is used when GCR_EL1.RRND == '1' to generate random
// Allocation Tags.
//
// The resulting Allocation Tag is selected from the set [0,15], excluding any Allocation Tag where
// exclude[tag_value] == 1. If 'exclude' is all Ones, the returned Allocation Tag is '0000'.
//
// This function is permitted to generate a non-deterministic selection from the set of non-excluded
// Allocation Tags. A reasonable implementation is described by the Pseudocode used when
// GCR_EL1.RRND is 0, but with a non-deterministic implementation of NextRandomTagBit(). Implementations
// may choose to behave the same as GCR_EL1.RRND=0.
//
// This function can read RGSR_EL1 and/or write RGSR_EL1 to an IMPLEMENTATION DEFINED value.
// If it is not capable of writing RGSR_EL1.SEED[15:0] to zero from a previous non-zero
// RGSR_EL1.SEED value, it is IMPLEMENTATION DEFINED whether the randomness is significantly
// impacted if RGSR_EL1.SEED[15:0] is set to zero.
bits(4) ChooseRandomNonExcludedTag(bits(16) exclude_in);
```

Library pseudocode for aarch64/functions/system/InGuardedPage

```
boolean InGuardedPage;
```

Library pseudocode for aarch64/functions/system/IsHCRXEL2Enabled

```
// IsHCRXEL2Enabled()
// ==================
// Returns TRUE if access to HCRX_EL2 register is enabled, and FALSE otherwise.
// Indirect read of HCRX_EL2 returns 0 when access is not enabled.

boolean IsHCRXEL2Enabled()
    assert(HaveFeatHCX());
    if HaveEL(EL3) && SCR_EL3.HXEn == '0' then
        return FALSE;

    return EL2Enabled();
```

Library pseudocode for aarch64/functions/system/SetBTypeCompatible

```
// SetBTypeCompatible()
// ====================
// Sets the value of BTypeCompatible global variable used by BTI

SetBTypeCompatible(boolean x)
    BTypeCompatible = x;
```

Library pseudocode for aarch64/functions/system/SetBTypeNext

```
// SetBTypeNext()
// ==============
// Set the value of BTypeNext global variable used by BTI

SetBTypeNext(bits(2) x)
    BTypeNext = x;
```

Library pseudocode for aarch64/functions/system/SetInGuardedPage

```
// SetInGuardedPage()
// ==================
// Global state updated to denote if memory access is from a guarded page.

SetInGuardedPage(boolean guardedpage)
    InGuardedPage = guardedpage;
```

Library pseudocode for aarch64/functions/tme/CheckTMEEnabled

```
// CheckTMEEnabled()
// =================
// Returns TRUE if access to TME instruction is enabled, FALSE otherwise.

CheckTMEEnabled()
    if PSTATE.EL IN {EL0, EL1, EL2} && HaveEL(EL3) then
        if SCR_EL3.TME == '0' then UNDEFINED;
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        if HCR_EL2.TME == '0' then UNDEFINED;
    return;
```

Library pseudocode for aarch64/functions/tme/CheckTransactionalSystemAccess

```
// CheckTransactionalSystemAccess()
// ================================
// Returns TRUE if an AArch64 MSR, MRS, or SYS instruction is permitted in
// Transactional state, based on the opcode's encoding, and FALSE otherwise.

boolean CheckTransactionalSystemAccess(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3) op2, b
    case read:op0:op1:crn:crm:op2 of
        when '0 00 011 0100 xxxx 11x' return TRUE;      // MSR (imm): DAIFSet, DAIFClr
        when '0 01 011 0111 0100 001' return TRUE;      // DC ZVA
        when '0 11 011 0100 0010 00x' return TRUE;      // MSR: NZCV, DAIF
        when '0 11 011 0100 0100 00x' return TRUE;      // MSR: FPCR, FPSR
        when '0 11 000 0100 0110 000' return TRUE;      // MSR: ICC_PMR_EL1
        when '0 11 011 1001 1100 100' return TRUE;      // MRS: PMSWINC_EL0
        when '1 11 xxx 0xxx xxxx xxx' return TRUE;      // MRS: op1=3, CRn=0..7
        when '1 11 xxx 100x xxxx xxx' return TRUE;      // MRS: op1=3, CRn=8..9
        when '1 11 xxx 1010 xxxx xxx' return TRUE;      // MRS: op1=3, CRn=10
        when '1 11 000 1100 1x00 010' return TRUE;      // MRS: op1=3, CRn=12 - ICC_HPPIRx_EL1
        when '1 11 000 1100 1011 011' return TRUE;      // MRS: op1=3, CRn=12 - ICC_RPR_EL1
        when '1 11 xxx 1101 xxxx xxx' return TRUE;      // MRS: op1=3, CRn=13
        when '1 11 xxx 1110 xxxx xxx' return TRUE;      // MRS: op1=3, CRn=14
        when '0 01 011 0111 0011 111' return TRUE;      // CPP RCTX
        when '0 01 011 0111 0011 10x' return TRUE;      // CFP RCTX, DVP RCTX
        when 'x 11 xxx 1x11 xxxx xxx' return boolean IMPLEMENTATION_DEFINED; // MRS: op1=3, CRn=11,15
        otherwise return FALSE;                         // all other SYS, SYSL, MRS, MSR
```

Library pseudocode for aarch64/functions/tme/CommitTransactionalWrites

```
// Makes all transactional writes to memory observable by other PEs and reset
// the transactional read and write sets.
CommitTransactionalWrites();
```

Library pseudocode for aarch64/functions/tme/DiscardTransactionalWrites

```
// Discards all transactional writes to memory and reset the transactional
// read and write sets.
DiscardTransactionalWrites();
```

Library pseudocode for aarch64/functions/tme/FailTransaction

```
// FailTransaction()
// =================

FailTransaction(TMFailure cause, boolean retry)
    FailTransaction(cause, retry, FALSE, Zeros(15));
    return;

// FailTransaction()
// =================
// Exits Transactional state and discards transactional updates to registers
// and memory.

FailTransaction(TMFailure cause, boolean retry, boolean interrupt, bits(15) reason)
    assert !retry || !interrupt;

    if HaveBRBExt() && BranchRecordAllowed(PSTATE.EL) then BRBFCR_EL1.LASTFAILED = '1';

    DiscardTransactionalWrites();
    // For trivial implementation no transaction checkpoint was taken
    if cause != TMFailure_TRIVIAL then
        RestoreTransactionCheckpoint();
    ClearExclusiveLocal(ProcessorID());

    bits(64) result = Zeros(64);

    result<23> = if interrupt then '1' else '0';
    result<15> = if retry && !interrupt then '1' else '0';
    case cause of
        when TMFailure_TRIVIAL result<24> = '1';
        when TMFailure_DBG     result<22> = '1';
        when TMFailure_NEST    result<21> = '1';
        when TMFailure_SIZE    result<20> = '1';
        when TMFailure_ERR     result<19> = '1';
        when TMFailure_IMP     result<18> = '1';
        when TMFailure_MEM     result<17> = '1';
        when TMFailure_CNCL    result<16> = '1'; result<14:0> = reason;

    TSTATE.depth = 0;
    X[TSTATE.Rt, 64] = result;
    boolean branch_conditional = FALSE;
    BranchTo(TSTATE.nPC, BranchType_TMFAIL, branch_conditional);
    EndOfInstruction();
    return;
```

Library pseudocode for aarch64/functions/tme/MemHasTransactionalAccess

```
// MemHasTransactionalAccess()
// ===========================
// Function checks if transactional accesses are not supported for an address
// range or memory type.

boolean MemHasTransactionalAccess(MemoryAttributes memattrs)
    if ((memattrs.shareability == Shareability_ISH ||
            memattrs.shareability == Shareability_OSH) &&
            memattrs.memtype == MemType_Normal &&
            memattrs.inner.attrs == MemAttr_WB &&
            memattrs.inner.hints == MemHint_RWA &&
            memattrs.inner.transient == FALSE &&
            memattrs.outer.hints == MemHint_RWA &&
            memattrs.outer.attrs == MemAttr_WB &&
            memattrs.outer.transient == FALSE) then
        return TRUE;
    else
        return boolean IMPLEMENTATION_DEFINED "Memory Region does not support Transactional access";
```

Library pseudocode for aarch64/functions/tme/RestoreTransactionCheckpoint

```
// RestoreTransactionCheckpoint()
// ==============================
// Restores part of the PE registers from the transaction checkpoint.

RestoreTransactionCheckpoint()
    SP[]             = TSTATE.SP;
    ICC_PMR_EL1      = TSTATE.ICC_PMR_EL1;
    PSTATE.<N,Z,C,V> = TSTATE.nzcv;
    PSTATE.<D,A,I,F> = TSTATE.<D,A,I,F>;

    for n = 0 to 30
        X[n, 64] = TSTATE.X[n];

    constant integer VL = CurrentVL;
    constant integer PL = VL DIV 8;
    if IsFPEnabled(PSTATE.EL) then
        if IsSVEEnabled(PSTATE.EL) then
            for n = 0 to 31
                Z[n, VL] = TSTATE.Z[n]<VL-1:0>;
            for n = 0 to 15
                P[n, PL] = TSTATE.P[n]<PL-1:0>;
            FFR[PL] = TSTATE.FFR<PL-1:0>;
        else
            for n = 0 to 31
                V[n, 128] = TSTATE.Z[n]<127:0>;
        FPCR = TSTATE.FPCR;
        FPSR = TSTATE.FPSR;

    return;
```

Library pseudocode for aarch64/functions/tme/StartTrackingTransactionalReadsWrites

```
// Starts tracking transactional reads and writes to memory.
StartTrackingTransactionalReadsWrites();
```

Library pseudocode for aarch64/functions/tme/TMFailure

```
enumeration TMFailure {
    TMFailure_CNCL,    // Executed a TCANCEL instruction
    TMFailure_DBG,     // A debug event was generated
    TMFailure_ERR,     // A non-permissible operation was attempted
    TMFailure_NEST,    // The maximum transactional nesting level was exceeded
    TMFailure_SIZE,    // The transactional read or write set limit was exceeded
    TMFailure_MEM,     // A transactional conflict occurred
    TMFailure_TRIVIAL, // Only a TRIVIAL version of TM is available
    TMFailure_IMP      // Any other failure cause
};
```

Library pseudocode for aarch64/functions/tme/TMState

```
type TMState is (
    integer       depth,              // Transaction nesting depth
    integer       Rt,                 // TSTART destination register
    bits(64)      nPC,                // Fallback instruction address
    array[0..30] of bits(64)     X,   // General purpose registers
    array[0..31] of bits(MAX_VL) Z,   // Vector registers
    array[0..15] of bits(MAX_PL) P,   // Predicate registers
    bits(MAX_PL)  FFR,                // First Fault Register
    bits(64)      SP,                 // Stack Pointer at current EL
    bits(64)      FPCR,               // Floating-point Control Register
    bits(64)      FPSR,               // Floating-point Status Register
    bits(64)      ICC_PMR_EL1,        // Interrupt Controller Interrupt Priority Mask Register
    bits(4)       nzcv,               // Condition flags
    bits(1)       D,                  // Debug mask bit
    bits(1)       A,                  // SError interrupt mask bit
    bits(1)       I,                  // IRQ mask bit
    bits(1)       F,                  // FIQ mask bit
)
```

Library pseudocode for aarch64/functions/tme/TSTATE

```
TMState TSTATE;
```

Library pseudocode for aarch64/functions/tme/TakeTransactionCheckpoint

```
// TakeTransactionCheckpoint()
// ===========================
// Captures part of the PE registers into the transaction checkpoint.

TakeTransactionCheckpoint()
    TSTATE.SP             = SP[];
    TSTATE.ICC_PMR_EL1    = ICC_PMR_EL1;
    TSTATE.nzcv           = PSTATE.<N,Z,C,V>;
    TSTATE.<D,A,I,F>      = PSTATE.<D,A,I,F>;

    for n = 0 to 30
        TSTATE.X[n] = X[n, 64];

    constant integer VL = CurrentVL;
    constant integer PL = VL DIV 8;
    if IsFPEnabled(PSTATE.EL) then
        if IsSVEEnabled(PSTATE.EL) then
            for n = 0 to 31
                TSTATE.Z[n]<VL-1:0> = Z[n, VL];
            for n = 0 to 15
                TSTATE.P[n]<PL-1:0> = P[n, PL];
            TSTATE.FFR<PL-1:0> = FFR[PL];
        else
            for n = 0 to 31
                TSTATE.Z[n]<127:0> = V[n, 128];
        TSTATE.FPCR = FPCR;
        TSTATE.FPSR = FPSR;

    return;
```

Library pseudocode for aarch64/functions/tme/TransactionStartTrap

```
// TransactionStartTrap()
// ======================
// Traps the execution of TSTART instruction.

TransactionStartTrap(integer dreg)
    bits(2) targetEL;
    bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_TSTARTAccessTrap);
    exception.syndrome<9:5> = dreg<4:0>;

    if UInt(PSTATE.EL) > UInt(EL1) then
        targetEL = PSTATE.EL;
    elsif EL2Enabled() && HCR_EL2.TGE == '1' then
        targetEL = EL2;
    else
        targetEL = EL1;
    AArch64.TakeException(targetEL, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/instrs/branch/eret/AArch64.ExceptionReturn

```
// AArch64.ExceptionReturn()
// =========================

AArch64.ExceptionReturn(bits(64) new_pc_in, bits(64) spsr)
    bits(64) new_pc = new_pc_in;
    if HaveTME() && TSTATE.depth > 0 then
        FailTransaction(TMFailure_ERR, FALSE);

    if HaveIESB() then
        sync_errors = SCTLR[].IESB == '1';
        if HaveDoubleFaultExt() then
            sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' && PSTATE.EL == EL3);
        if sync_errors then
            SynchronizeErrors();
            iesb_req = TRUE;
            TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
    SynchronizeContext();

    // Attempts to change to an illegal state will invoke the Illegal Execution state mechanism
    bits(2) source_el = PSTATE.EL;
    boolean illegal_psr_state = IllegalExceptionReturn(spsr);
    SetPSTATEFromPSR(spsr, illegal_psr_state);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();

    if illegal_psr_state && spsr<4> == '1' then
        // If the exception return is illegal, PC[63:32,1:0] are UNKNOWN
        new_pc<63:32> = bits(32) UNKNOWN;
        new_pc<1:0> = bits(2) UNKNOWN;
    elsif UsingAArch32() then               // Return to AArch32
        // ELR_ELx[1:0] or ELR_ELx[0] are treated as being 0, depending on the
        // target instruction set state
        if PSTATE.T == '1' then
            new_pc<0> = '0';                // T32
        else
            new_pc<1:0> = '00';             // A32
    else                                    // Return to AArch64
        // ELR_ELx[63:56] might include a tag
        new_pc = AArch64.BranchAddr(new_pc, PSTATE.EL);

    if HaveBRBExt() then
        BRBEExceptionReturn(new_pc, source_el);

    if UsingAArch32() then
        if HaveSME() && PSTATE.SM == '1' then ResetSVEState();

        // 32 most significant bits are ignored.
        boolean branch_conditional = FALSE;
        BranchTo(new_pc<31:0>, BranchType_ERET, branch_conditional);
    else
        BranchToAddr(new_pc, BranchType_ERET);

    CheckExceptionCatch(FALSE);             // Check for debug event on exception return
```

Library pseudocode for aarch64/instrs/countop/CountOp

```
enumeration CountOp     {CountOp_CLZ, CountOp_CLS, CountOp_CNT};
```

Library pseudocode for aarch64/instrs/extendreg/DecodeRegExtend

```
// DecodeRegExtend()
// =================
// Decode a register extension option

ExtendType DecodeRegExtend(bits(3) op)
    case op of
        when '000' return ExtendType_UXTB;
        when '001' return ExtendType_UXTH;
        when '010' return ExtendType_UXTW;
        when '011' return ExtendType_UXTX;
        when '100' return ExtendType_SXTB;
        when '101' return ExtendType_SXTH;
        when '110' return ExtendType_SXTW;
        when '111' return ExtendType_SXTX;
```

Library pseudocode for aarch64/instrs/extendreg/ExtendReg

```
// ExtendReg()
// ===========
// Perform a register extension and shift

bits(N) ExtendReg(integer reg, ExtendType exttype, integer shift, integer N)
    assert shift >= 0 && shift <= 4;
    bits(N) val = X[reg, N];
    boolean unsigned;
    integer len;

    case exttype of
        when ExtendType_SXTB unsigned = FALSE; len = 8;
        when ExtendType_SXTH unsigned = FALSE; len = 16;
        when ExtendType_SXTW unsigned = FALSE; len = 32;
        when ExtendType_SXTX unsigned = FALSE; len = 64;
        when ExtendType_UXTB unsigned = TRUE;  len = 8;
        when ExtendType_UXTH unsigned = TRUE;  len = 16;
        when ExtendType_UXTW unsigned = TRUE;  len = 32;
        when ExtendType_UXTX unsigned = TRUE;  len = 64;

    // Note the extended width of the intermediate value and
    // that sign extension occurs from bit <len+shift-1>, not
    // from bit <len-1>. This is equivalent to the instruction
    //   [SU]BFIZ Rtmp, Rreg, #shift, #len
    // It may also be seen as a sign/zero extend followed by a shift:
    //   LSL(Extend(val<len-1:0>, N, unsigned), shift);

    len = Min(len, N - shift);
    return Extend(val<len-1:0> : Zeros(shift), N, unsigned);
```

Library pseudocode for aarch64/instrs/extendreg/ExtendType

```
enumeration ExtendType  {ExtendType_SXTB, ExtendType_SXTH, ExtendType_SXTW, ExtendType_SXTX,
                         ExtendType_UXTB, ExtendType_UXTH, ExtendType_UXTW, ExtendType_UXTX};
```

Library pseudocode for aarch64/instrs/float/arithmetic/max-min/fpmaxminop/FPMaxMinOp

```
enumeration FPMaxMinOp  {FPMaxMinOp_MAX, FPMaxMinOp_MIN,
                         FPMaxMinOp_MAXNUM, FPMaxMinOp_MINNUM};
```

Library pseudocode for aarch64/instrs/float/arithmetic/unary/fpunaryop/FPUnaryOp

```
enumeration FPUnaryOp   {FPUnaryOp_ABS, FPUnaryOp_MOV,
                         FPUnaryOp_NEG, FPUnaryOp_SQRT};
```

Library pseudocode for aarch64/instrs/float/convert/fpconvop/FPConvOp

```
enumeration FPConvOp    {FPConvOp_CVT_FtoI, FPConvOp_CVT_ItoF,
                         FPConvOp_MOV_FtoI, FPConvOp_MOV_ItoF
                         , FPConvOp_CVT_FtoI_JS
};
```

Library pseudocode for aarch64/instrs/integer/bitfield/bfxpreferred/BFXPreferred

```
// BFXPreferred()
// ==============
//
// Return TRUE if UBFX or SBFX is the preferred disassembly of a
// UBFM or SBFM bitfield instruction. Must exclude more specific
// aliases UBFIZ, SBFIZ, UXT[BH], SXT[BHW], LSL, LSR and ASR.

boolean BFXPreferred(bit sf, bit uns, bits(6) imms, bits(6) immr)

    // must not match UBFIZ/SBFIX alias
    if UInt(imms) < UInt(immr) then
        return FALSE;

    // must not match LSR/ASR/LSL alias (imms == 31 or 63)
    if imms == sf:'11111' then
        return FALSE;

    // must not match UXTx/SXTx alias
    if immr == '000000' then
        // must not match 32-bit UXT[BH] or SXT[BH]
        if sf == '0' && imms IN {'000111', '001111'} then
            return FALSE;
        // must not match 64-bit SXT[BHW]
        if sf:uns == '10' && imms IN {'000111', '001111', '011111'} then
            return FALSE;

    // must be UBFX/SBFX alias
    return TRUE;
```

Library pseudocode for aarch64/instrs/integer/bitmasks/AltDecodeBitMasks

```
// AltDecodeBitMasks()
// ===================
// Alternative but logically equivalent implementation of DecodeBitMasks() that
// uses simpler primitives to compute tmask and wmask.

(bits(M), bits(M)) AltDecodeBitMasks(bit immN, bits(6) imms, bits(6) immr,
                                     boolean immediate, integer M)
    bits(64) tmask, wmask;
    bits(6) tmask_and, wmask_and;
    bits(6) tmask_or, wmask_or;
    bits(6) levels;

    // Compute log2 of element size
    // 2^len must be in range [2, M]
    len = HighestSetBit(immN:NOT(imms));
    if len < 1 then UNDEFINED;
    assert M >= (1 << len);

    // Determine s, r and s - r parameters
    levels = ZeroExtend(Ones(len), 6);

    // For logical immediates an all-ones value of s is reserved
    // since it would generate a useless all-ones result (many times)
    if immediate && (imms AND levels) == levels then
        UNDEFINED;

    s = UInt(imms AND levels);
    r = UInt(immr AND levels);
    diff = s - r;    // 6-bit subtract with borrow

    // Compute "top mask"
    tmask_and = diff<5:0> OR NOT(levels);
    tmask_or  = diff<5:0> AND levels;

    tmask = Ones(64);
    tmask = ((tmask
              AND Replicate(Replicate(tmask_and<0>, 1) : Ones(1), 32))
               OR  Replicate(Zeros(1) : Replicate(tmask_or<0>, 1), 32));
    // optimization of first step:
    // tmask = Replicate(tmask_and<0> : '1', 32);
    tmask = ((tmask
              AND Replicate(Replicate(tmask_and<1>, 2) : Ones(2), 16))
               OR  Replicate(Zeros(2) : Replicate(tmask_or<1>, 2), 16));
    tmask = ((tmask
              AND Replicate(Replicate(tmask_and<2>, 4) : Ones(4), 8))
               OR  Replicate(Zeros(4) : Replicate(tmask_or<2>, 4), 8));
    tmask = ((tmask
              AND Replicate(Replicate(tmask_and<3>, 8) : Ones(8), 4))
               OR  Replicate(Zeros(8) : Replicate(tmask_or<3>, 8), 4));
    tmask = ((tmask
              AND Replicate(Replicate(tmask_and<4>, 16) : Ones(16), 2))
               OR  Replicate(Zeros(16) : Replicate(tmask_or<4>, 16), 2));
    tmask = ((tmask
              AND Replicate(Replicate(tmask_and<5>, 32) : Ones(32), 1))
               OR  Replicate(Zeros(32) : Replicate(tmask_or<5>, 32), 1));

    // Compute "wraparound mask"
    wmask_and = immr OR NOT(levels);
    wmask_or  = immr AND levels;

    wmask = Zeros(64);
    wmask = ((wmask
              AND Replicate(Ones(1) : Replicate(wmask_and<0>, 1), 32))
               OR  Replicate(Replicate(wmask_or<0>, 1) : Zeros(1), 32));
    // optimization of first step:
    // wmask = Replicate(wmask_or<0> : '0', 32);
    wmask = ((wmask
              AND Replicate(Ones(2) : Replicate(wmask_and<1>, 2), 16))
               OR  Replicate(Replicate(wmask_or<1>, 2) : Zeros(2), 16));
    wmask = ((wmask
```

```
            AND Replicate(Ones(4) : Replicate(wmask_and<2>, 4), 8))
             OR  Replicate(Replicate(wmask_or<2>, 4) : Zeros(4), 8));
    wmask = ((wmask
            AND Replicate(Ones(8) : Replicate(wmask_and<3>, 8), 4))
             OR  Replicate(Replicate(wmask_or<3>, 8) : Zeros(8), 4));
    wmask = ((wmask
            AND Replicate(Ones(16) : Replicate(wmask_and<4>, 16), 2))
             OR  Replicate(Replicate(wmask_or<4>, 16) : Zeros(16), 2));
    wmask = ((wmask
            AND Replicate(Ones(32) : Replicate(wmask_and<5>, 32), 1))
             OR  Replicate(Replicate(wmask_or<5>, 32) : Zeros(32), 1));

    if diff<6> != '0' then // borrow from s - r
        wmask = wmask AND tmask;
    else
        wmask = wmask OR tmask;

    return (wmask<M-1:0>, tmask<M-1:0>);
```

Library pseudocode for aarch64/instrs/integer/bitmasks/DecodeBitMasks

```
// DecodeBitMasks()
// ================
// Decode AArch64 bitfield and logical immediate masks which use a similar encoding structure

(bits(M), bits(M)) DecodeBitMasks(bit immN, bits(6) imms, bits(6) immr,
                                  boolean immediate, integer M)
    bits(M) tmask, wmask;
    bits(6) levels;

    // Compute log2 of element size
    // 2^len must be in range [2, M]
    len = HighestSetBit(immN:NOT(imms));
    if len < 1 then UNDEFINED;
    assert M >= (1 << len);

    // Determine s, r and s - r parameters
    levels = ZeroExtend(Ones(len), 6);

    // For logical immediates an all-ones value of s is reserved
    // since it would generate a useless all-ones result (many times)
    if immediate && (imms AND levels) == levels then
        UNDEFINED;

    s = UInt(imms AND levels);
    r = UInt(immr AND levels);
    diff = s - r;    // 6-bit subtract with borrow

    esize = 1 << len;
    d = UInt(diff<len-1:0>);
    welem = ZeroExtend(Ones(s + 1), esize);
    telem = ZeroExtend(Ones(d + 1), esize);
    wmask = Replicate(ROR(welem, r));
    tmask = Replicate(telem);
    return (wmask, tmask);
```

Library pseudocode for aarch64/instrs/integer/ins-ext/insert/movewide/movewideop/MoveWideOp

```
enumeration MoveWideOp  {MoveWideOp_N, MoveWideOp_Z, MoveWideOp_K};
```

Library pseudocode for aarch64/instrs/integer/logical/movwpreferred/MoveWidePreferred

```
// MoveWidePreferred()
// ===================
//
// Return TRUE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single MOVZ or MOVN instruction.
// Used as a condition for the preferred MOV<-ORR alias.

boolean MoveWidePreferred(bit sf, bit immN, bits(6) imms, bits(6) immr)
    integer s = UInt(imms);
    integer r = UInt(immr);
    integer width = if sf == '1' then 64 else 32;

    // element size must equal total immediate size
    if sf == '1' && !((immN:imms) IN {'1xxxxxx'}) then
        return FALSE;
    if sf == '0' && !((immN:imms) IN {'00xxxxx'}) then
        return FALSE;

    // for MOVZ must contain no more than 16 ones
    if s < 16 then
        // ones must not span halfword boundary when rotated
        return (-r MOD 16) <= (15 - s);

    // for MOVN must contain no more than 16 zeros
    if s >= width - 15 then
        // zeros must not span halfword boundary when rotated
        return (r MOD 16) <= (s - (width - 15));

    return FALSE;
```

Library pseudocode for aarch64/instrs/integer/shiftreg/DecodeShift

```
// DecodeShift()
// =============
// Decode shift encodings

ShiftType DecodeShift(bits(2) op)
    case op of
        when '00'  return ShiftType_LSL;
        when '01'  return ShiftType_LSR;
        when '10'  return ShiftType_ASR;
        when '11'  return ShiftType_ROR;
```

Library pseudocode for aarch64/instrs/integer/shiftreg/ShiftReg

```
// ShiftReg()
// ==========
// Perform shift of a register operand

bits(N) ShiftReg(integer reg, ShiftType shiftype, integer amount, integer N)
    bits(N) result = X[reg, N];
    case shiftype of
        when ShiftType_LSL result = LSL(result, amount);
        when ShiftType_LSR result = LSR(result, amount);
        when ShiftType_ASR result = ASR(result, amount);
        when ShiftType_ROR result = ROR(result, amount);
    return result;
```

Library pseudocode for aarch64/instrs/integer/shiftreg/ShiftType

```
enumeration ShiftType   {ShiftType_LSL, ShiftType_LSR, ShiftType_ASR, ShiftType_ROR};
```

Library pseudocode for aarch64/instrs/logicalop/LogicalOp

```
enumeration LogicalOp   {LogicalOp_AND, LogicalOp_EOR, LogicalOp_ORR};
```

Library pseudocode for aarch64/instrs/memory/memop/MemAtomicOp

```
enumeration MemAtomicOp {MemAtomicOp_ADD,
                         MemAtomicOp_BIC,
                         MemAtomicOp_EOR,
                         MemAtomicOp_ORR,
                         MemAtomicOp_SMAX,
                         MemAtomicOp_SMIN,
                         MemAtomicOp_UMAX,
                         MemAtomicOp_UMIN,
                         MemAtomicOp_SWP};
```

Library pseudocode for aarch64/instrs/memory/memop/MemOp

```
enumeration MemOp {MemOp_LOAD, MemOp_STORE, MemOp_PREFETCH};
```

Library pseudocode for aarch64/instrs/memory/prefetch/Prefetch

```
// Prefetch()
// ==========

// Decode and execute the prefetch hint on ADDRESS specified by PRFOP

Prefetch(bits(64) address, bits(5) prfop)
    PrefetchHint hint;
    integer target;
    boolean stream;

    case prfop<4:3> of
        when '00' hint = Prefetch_READ;         // PLD: prefetch for load
        when '01' hint = Prefetch_EXEC;         // PLI: preload instructions
        when '10' hint = Prefetch_WRITE;        // PST: prepare for store
        when '11' return;                       // unallocated hint
    target = UInt(prfop<2:1>);                  // target cache level
    stream = (prfop<0> != '0');                 // streaming (non-temporal)
    Hint_Prefetch(address, hint, target, stream);
    return;
```

Library pseudocode for aarch64/instrs/system/barriers/barrierop/MemBarrierOp

```
enumeration MemBarrierOp   {  MemBarrierOp_DSB         // Data Synchronization Barrier
                            , MemBarrierOp_DMB         // Data Memory Barrier
                            , MemBarrierOp_ISB         // Instruction Synchronization Barrier
                            , MemBarrierOp_SSBB        // Speculative Synchronization Barrier to VA
                            , MemBarrierOp_PSSBB       // Speculative Synchronization Barrier to PA
                            , MemBarrierOp_SB          // Speculation Barrier
                           };
```

Library pseudocode for aarch64/instrs/system/hints/syshintop/SystemHintOp

```
enumeration SystemHintOp {
    SystemHintOp_NOP,
    SystemHintOp_YIELD,
    SystemHintOp_WFE,
    SystemHintOp_WFI,
    SystemHintOp_SEV,
    SystemHintOp_SEVL,
    SystemHintOp_DGH,
    SystemHintOp_ESB,
    SystemHintOp_PSB,
    SystemHintOp_TSB,
    SystemHintOp_BTI,
    SystemHintOp_WFET,
    SystemHintOp_WFIT,
    SystemHintOp_CSDB
};
```

Library pseudocode for aarch64/instrs/system/register/cpsr/pstatefield/PSTATEField

```
enumeration PSTATEField {PSTATEField_DAIFSet, PSTATEField_DAIFClr,
                         PSTATEField_PAN, // Armv8.1
                         PSTATEField_UAO, // Armv8.2
                         PSTATEField_DIT, // Armv8.4
                         PSTATEField_SSBS,
                         PSTATEField_TCO, // Armv8.5
                         PSTATEField_SVCRSM,
                         PSTATEField_SVCRZA,
                         PSTATEField_SVCRSMZA,
                         PSTATEField_ALLINT,
                         PSTATEField_SP
                         };
```

Library pseudocode for aarch64/instrs/system/sysops/at/AArch64.AT

```
// AArch64.AT()
// ============
// Perform address translation as per AT instructions.

AArch64.AT(bits(64) address, TranslationStage stage_in, bits(2) el_in, ATAccess ataccess)
    TranslationStage stage = stage_in;
    bits(2) el = el_in;
    if HaveRME() && PSTATE.EL == EL3 && SCR_EL3.<NSE,NS> == '10' && el != EL3 then UNDEFINED;
    // For stage 1 translation, when HCR_EL2.{E2H, TGE} is {1,1} and requested EL is EL1,
    // the EL2&0 translation regime is used.
    if HCR_EL2.<E2H, TGE> == '11' && el == EL1 && stage == TranslationStage_1 then
        el = EL2;
    if HaveEL(EL3) && stage == TranslationStage_12 && !EL2Enabled() then
        stage = TranslationStage_1;

    acctype = if ataccess IN {ATAccess_Read, ATAccess_Write} then AccType_AT else AccType_ATPAN;
    iswrite = ataccess IN {ATAccess_WritePAN, ATAccess_Write};
    aligned = TRUE;
    ispriv  = el != EL0;

    fault = NoFault();
    fault.acctype = acctype;
    fault.write = iswrite;

    Regime regime;
    if stage == TranslationStage_12 then
        regime = Regime_EL10;
    else
        regime = TranslationRegime(el, acctype);

    AddressDescriptor addrdesc;
    ss = SecurityStateAtEL(el);
    if (el == EL0 && ELUsingAArch32(EL1)) || (el != EL0 && ELUsingAArch32(el)) then
        if regime == Regime_EL2 || TTBCR.EAE == '1' then
            (fault, addrdesc) = AArch32.S1TranslateLD(fault, regime, ss, address<31:0>, acctype,
                                                      aligned, iswrite, ispriv);
        else
            (fault, addrdesc, -) = AArch32.S1TranslateSD(fault, regime, ss, address<31:0>, acctype,
                                                         aligned, iswrite, ispriv);
    else
        (fault, addrdesc) = AArch64.S1Translate(fault, regime, ss, address, acctype, aligned,
                                                iswrite, ispriv);

    if stage == TranslationStage_12 && fault.statuscode == Fault_None then
        boolean s2fs1walk;
        boolean s1aarch64;
        if ELUsingAArch32(EL1) && regime == Regime_EL10 && EL2Enabled() then
            addrdesc.vaddress = ZeroExtend(address, 64);
            s2fs1walk = FALSE;
            (fault, addrdesc) = AArch32.S2Translate(fault, addrdesc, ss, s2fs1walk, acctype,
                                                    aligned, iswrite, ispriv);
        elsif regime == Regime_EL10 && EL2Enabled() then
            s1aarch64 = TRUE;
            s2fs1walk = FALSE;
            (fault, addrdesc) = AArch64.S2Translate(fault, addrdesc, s1aarch64, ss, s2fs1walk,
                                                    acctype, aligned, iswrite, ispriv);

    is_ATS1Ex = stage != TranslationStage_12;
    if fault.statuscode != Fault_None then
        addrdesc = CreateFaultyAddressDescriptor(address, fault);
        // Take an exception on:
        // * A Synchronous External abort occurs on translation table walk
        // * A stage 2 fault occurs on a stage 1 walk
        // * A GPC Exception (FEAT_RME)
        // * A GPF from ATS1E{1,0}* when executed from EL1 and HCR_EL2.GPF == '1' (FEAT_RME)
        if (IsExternalAbort(fault) ||
                (PSTATE.EL == EL1 && fault.s2fs1walk) ||
                (HaveRME() && fault.gpcf.gpf != GPCF_None && (
                    ReportAsGPCException(fault) ||
                    (HCR_EL2.GPF == '1' && PSTATE.EL == EL1 && el IN {EL1, EL0} && is_ATS1Ex)
```

```
            ))) then
            PAR_EL1 = bits(64) UNKNOWN;
            AArch64.Abort(address, addrdesc.fault);

    AArch64.EncodePAR(regime, is_ATS1Ex, addrdesc);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/at/AArch64.EncodePAR

```
// AArch64.EncodePAR()
// ===================
// Encode PAR register with result of translation.

AArch64.EncodePAR(Regime regime, boolean is_ATS1Ex, AddressDescriptor addrdesc)
    PAR_EL1 = Zeros(64);
    paspace = addrdesc.paddress.paspace;

    if !IsFault(addrdesc) then
        PAR_EL1.F = '0';
        if HaveRME() then
            if regime == Regime_EL3 then
                case paspace of
                    when PAS_Secure     PAR_EL1.<NSE,NS> = '00';
                    when PAS_NonSecure  PAR_EL1.<NSE,NS> = '01';
                    when PAS_Root       PAR_EL1.<NSE,NS> = '10';
                    when PAS_Realm      PAR_EL1.<NSE,NS> = '11';

            elsif SecurityStateForRegime(regime) == SS_Secure  then
                PAR_EL1.NSE = bit UNKNOWN;
                PAR_EL1.NS = if paspace == PAS_Secure then '0' else '1';

            elsif SecurityStateForRegime(regime) == SS_Realm then
                if regime == Regime_EL10 && is_ATS1Ex then
                    PAR_EL1.NSE = bit UNKNOWN;
                    PAR_EL1.NS  = bit UNKNOWN;
                else
                    PAR_EL1.NSE = bit UNKNOWN;
                    PAR_EL1.NS  = if paspace == PAS_Realm then '0' else '1';

            else
                PAR_EL1.NSE = bit UNKNOWN;
                PAR_EL1.NS  = bit UNKNOWN;
        else
            PAR_EL1<11> = '1'; // RES1
            if SecurityStateForRegime(regime) == SS_Secure then
                PAR_EL1.NS = if paspace == PAS_Secure then '0' else '1';
            else
                PAR_EL1.NS = bit UNKNOWN;
        PAR_EL1.SH   = ReportedPARShareability(PAREncodeShareability(addrdesc.memattrs));
        PAR_EL1.PA   = addrdesc.paddress.address<52-1:12>;
        PAR_EL1.ATTR = ReportedPARAttrs(EncodePARAttrs(addrdesc.memattrs));
        PAR_EL1<10> = bit IMPLEMENTATION_DEFINED "Non-Faulting PAR";
    else
        PAR_EL1.F   = '1';
        PAR_EL1.FST = AArch64.PARFaultStatus(addrdesc.fault);
        PAR_EL1.PTW = if addrdesc.fault.s2fs1walk then '1' else '0';
        PAR_EL1.S   = if addrdesc.fault.secondstage then '1' else '0';
        PAR_EL1<11> = '1'; // RES1
        PAR_EL1<63:48> = bits(16) IMPLEMENTATION_DEFINED "Faulting PAR";
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/at/AArch64.PARFaultStatus

```
// AArch64.PARFaultStatus()
// ========================
// Fault status field decoding of 64-bit PAR.

bits(6) AArch64.PARFaultStatus(FaultRecord fault)
    bits(6) fst;

    if fault.statuscode == Fault_Domain then
        // Report Domain fault
        assert fault.level IN {1,2};
        fst<1:0> = if fault.level == 1 then '01' else '10';
        fst<5:2> = '1111';
    else
        fst = EncodeLDFSC(fault.statuscode, fault.level);
    return fst;
```

Library pseudocode for aarch64/instrs/system/sysops/dc/AArch64.DC

```
// AArch64.DC()
// ============
// Perform Data Cache Operation.

AArch64.DC(bits(64) regval, CacheType cachetype, CacheOp cacheop, CacheOpScope opscope_in)
    CacheOpScope opscope = opscope_in;
    AccType acctype = AccType_DC;
    CacheRecord cache;

    cache.acctype = acctype;
    cache.cachetype = cachetype;
    cache.cacheop = cacheop;
    cache.opscope = opscope;

    if opscope == CacheOpScope_SetWay then
        ss = SecurityStateAtEL(PSTATE.EL);
        cache.cpas = CPASAtSecurityState(ss);
        cache.shareability = Shareability_NSH;
        (cache.set, cache.way, cache.level) = DecodeSW(regval, cachetype);
        if (cacheop == CacheOp_Invalidate && PSTATE.EL == EL1 && EL2Enabled() &&
          (HCR_EL2.SWIO == '1' || HCR_EL2.<DC,VM> != '00')) then
            cache.cacheop = CacheOp_CleanInvalidate;

        CACHE_OP(cache);
        return;

    if EL2Enabled() && !IsInHost() then
        if PSTATE.EL IN {EL0, EL1} then
            cache.is_vmid_valid = TRUE;
            cache.vmid          = VMID[];
        else
            cache.is_vmid_valid = FALSE;
    else
        cache.is_vmid_valid = FALSE;

    if PSTATE.EL == EL0 then
        cache.is_asid_valid = TRUE;
        cache.asid          = ASID[];
    else
        cache.is_asid_valid = FALSE;

    if opscope == CacheOpScope_PoDP && boolean IMPLEMENTATION_DEFINED "Memory system does not supports Po
        opscope = CacheOpScope_PoP;
    if opscope == CacheOpScope_PoP && boolean IMPLEMENTATION_DEFINED "Memory system does not supports PoP
        opscope = CacheOpScope_PoC;
    need_translate = DCInstNeedsTranslation(opscope);
    iswrite = cacheop == CacheOp_Invalidate;
    vaddress = regval;

    size = 0;        // by default no watchpoint address
    if iswrite then
        size = integer IMPLEMENTATION_DEFINED "Data Cache Invalidate Watchpoint Size";
        assert size >= 4*(2^(UInt(CTR_EL0.DminLine))) && size <= 2048;
        assert UInt(size<32:0> AND (size-1)<32:0>) == 0; // size is power of 2
        vaddress = Align(regval, size);

    cache.translated = need_translate;
    cache.vaddress = vaddress;

    if need_translate then
        wasaligned = TRUE;
        memaddrdesc = AArch64.TranslateAddress(vaddress, acctype, iswrite,
                                               wasaligned, size);
        if IsFault(memaddrdesc) then
            AArch64.Abort(regval, memaddrdesc.fault);

        memattrs = memaddrdesc.memattrs;
        cache.paddress = memaddrdesc.paddress;
        cache.cpas = CPASAtPAS(memaddrdesc.paddress.paspace);
        if opscope IN {CacheOpScope_PoC, CacheOpScope_PoP, CacheOpScope_PoDP} then
```

```
            cache.shareability = memattrs.shareability;
        else
            cache.shareability = Shareability_NSH;
    else
        cache.shareability = Shareability UNKNOWN;
        cache.paddress = FullAddress UNKNOWN;

    if cacheop == CacheOp_Invalidate && PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.<DC,VM> != '00' then
        cache.cacheop = CacheOp_CleanInvalidate;

    CACHE_OP(cache);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/dc/AArch64.MemZero

```
// AArch64.MemZero()
// =================

AArch64.MemZero(bits(64) regval, CacheType cachetype)

    AccType acctype = AccType_DCZVA;
    boolean iswrite = TRUE;
    boolean wasaligned = TRUE;

    integer size = 4*(2^(UInt(DCZID_EL0.BS)));
    bits(64) vaddress = Align(regval, size);

    memaddrdesc = AArch64.TranslateAddress(vaddress, acctype, iswrite,
                                           wasaligned, size);

    if IsFault(memaddrdesc) then
        if IsDebugException(memaddrdesc.fault) then
            AArch64.Abort(vaddress, memaddrdesc.fault);
        else
            AArch64.Abort(regval, memaddrdesc.fault);
    else
        if cachetype == CacheType_Data then
            AArch64.DataMemZero(regval, vaddress, memaddrdesc, size);
        elsif cachetype == CacheType_Tag then
            if HaveMTEExt() then AArch64.TagMemZero(vaddress, size);
        elsif cachetype == CacheType_Data_Tag then
            if HaveMTEExt() then AArch64.TagMemZero(vaddress, size);
            AArch64.DataMemZero(regval, vaddress, memaddrdesc, size);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/ic/AArch64.IC

```
// AArch64.IC()
// ============
// Perform Instruction Cache Operation.

AArch64.IC(CacheOpScope opscope)
    regval = bits(64) UNKNOWN;
    AArch64.IC(regval, opscope);

// AArch64.IC()
// ============
// Perform Instruction Cache Operation.

AArch64.IC(bits(64) regval, CacheOpScope opscope)
    CacheRecord cache;
    AccType acctype = AccType_IC;

    cache.acctype = acctype;
    cache.cachetype = CacheType_Instruction;
    cache.cacheop = CacheOp_Invalidate;
    cache.opscope = opscope;

    if opscope IN {CacheOpScope_ALLU, CacheOpScope_ALLUIS} then
        ss = SecurityStateAtEL(PSTATE.EL);
        cache.cpas = CPASAtSecurityState(ss);
        if (opscope == CacheOpScope_ALLUIS || (opscope == CacheOpScope_ALLU && PSTATE.EL == EL1
            && EL2Enabled() && HCR_EL2.FB == '1')) then
            cache.shareability = Shareability_ISH;
        else
            cache.shareability = Shareability_NSH;
        cache.regval = regval;
        CACHE_OP(cache);
    else
        assert opscope == CacheOpScope_PoU;

        if EL2Enabled() && !IsInHost() then
            if PSTATE.EL IN {EL0, EL1} then
                cache.is_vmid_valid = TRUE;
                cache.vmid          = VMID[];
            else
                cache.is_vmid_valid = FALSE;
        else
            cache.is_vmid_valid = FALSE;

        if PSTATE.EL == EL0 then
            cache.is_asid_valid = TRUE;
            cache.asid          = ASID[];
        else
            cache.is_asid_valid = FALSE;

        bits(64) vaddress = regval;
        need_translate = ICInstNeedsTranslation(opscope);

        cache.vaddress = regval;
        cache.shareability = Shareability_NSH;
        cache.translated = need_translate;

        if !need_translate then
            cache.paddress = FullAddress UNKNOWN;
            CACHE_OP(cache);
            return;
        iswrite = FALSE;
        wasaligned = TRUE;
        size = 0;
        memaddrdesc = AArch64.TranslateAddress(vaddress, acctype, iswrite,
                                               wasaligned, size);

        if IsFault(memaddrdesc) then
            AArch64.Abort(regval, memaddrdesc.fault);

        cache.cpas = CPASAtPAS(memaddrdesc.paddress.paspace);
```

```
        cache.paddress = memaddrdesc.paddress;
        CACHE_OP(cache);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/predictionrestrict/RestrictPrediction

```
// RestrictPrediction()
// ====================
// Clear all predictions in the context.

AArch64.RestrictPrediction(bits(64) val, RestrictType restriction)

    ExecutionCntxt c;
    target_el    = val<25:24>;

    // If the instruction is executed at an EL lower than the specified
    // level, it is treated as a NOP.
    if UInt(target_el) > UInt(PSTATE.EL) then return;

    bit ns  = val<26>;
    bit nse = val<27>;
    ss = TargetSecurityState(ns, nse);
    if ss == SS_Root && target_el != EL3 then return;

    c.security  = ss;
    c.target_el = target_el;

    if EL2Enabled() && !IsInHost() then
        if PSTATE.EL IN {EL0, EL1} then
            c.is_vmid_valid = TRUE;
            c.all_vmid      = FALSE;
            c.vmid          = VMID[];

        elsif target_el IN {EL0, EL1} then
            c.is_vmid_valid = TRUE;
            c.all_vmid      = val<48> == '1';
            c.vmid          = val<47:32>;           // Only valid if  val<48> == '0';
        else
            c.is_vmid_valid = FALSE;
    else
        c.is_vmid_valid = FALSE;

    if PSTATE.EL == EL0 then
        c.is_asid_valid = TRUE;
        c.all_asid      = FALSE;
        c.asid          = ASID[];

    elsif target_el == EL0 then
        c.is_asid_valid = TRUE;
        c.all_asid      = val<16> == '1';
        c.asid          = val<15:0>;                  // Only valid if  val<16> == '0';

    else
        c.is_asid_valid = FALSE;

    c.restriction = restriction;
    RESTRICT_PREDICTIONS(c);
```

Library pseudocode for aarch64/instrs/system/sysops/sysop/SysOp

```
// SysOp()
// =======

SystemOp SysOp(bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2)
    case op1:CRn:CRm:op2 of
        when '000 0111 1000 000' return Sys_AT;    // S1E1R
        when '000 0111 1000 001' return Sys_AT;    // S1E1W
        when '000 0111 1000 010' return Sys_AT;    // S1E0R
        when '000 0111 1000 011' return Sys_AT;    // S1E0W
        when '000 0111 1001 000' return Sys_AT;    // S1E1RP
        when '000 0111 1001 001' return Sys_AT;    // S1E1WP
        when '100 0111 1000 000' return Sys_AT;    // S1E2R
        when '100 0111 1000 001' return Sys_AT;    // S1E2W
        when '100 0111 1000 100' return Sys_AT;    // S12E1R
        when '100 0111 1000 101' return Sys_AT;    // S12E1W
        when '100 0111 1000 110' return Sys_AT;    // S12E0R
        when '100 0111 1000 111' return Sys_AT;    // S12E0W
        when '110 0111 1000 000' return Sys_AT;    // S1E3R
        when '110 0111 1000 001' return Sys_AT;    // S1E3W
        when '001 0111 0010 100' return Sys_BRB;   // IALL
        when '001 0111 0010 101' return Sys_BRB;   // INJ
        when '000 0111 0110 001' return Sys_DC;    // IVAC
        when '000 0111 0110 010' return Sys_DC;    // ISW
        when '000 0111 0110 011' return Sys_DC;    // IGVAC
        when '000 0111 0110 100' return Sys_DC;    // IGSW
        when '000 0111 0110 101' return Sys_DC;    // IGDVAC
        when '000 0111 0110 110' return Sys_DC;    // IGDSW
        when '000 0111 1010 010' return Sys_DC;    // CSW
        when '000 0111 1010 100' return Sys_DC;    // CGSW
        when '000 0111 1010 110' return Sys_DC;    // CGDSW
        when '000 0111 1110 010' return Sys_DC;    // CISW
        when '000 0111 1110 100' return Sys_DC;    // CIGSW
        when '000 0111 1110 110' return Sys_DC;    // CIGDSW
        when '011 0111 0100 001' return Sys_DC;    // ZVA
        when '011 0111 0100 011' return Sys_DC;    // GVA
        when '011 0111 0100 100' return Sys_DC;    // GZVA
        when '011 0111 1010 001' return Sys_DC;    // CVAC
        when '011 0111 1010 011' return Sys_DC;    // CGVAC
        when '011 0111 1010 101' return Sys_DC;    // CGDVAC
        when '011 0111 1011 001' return Sys_DC;    // CVAU
        when '011 0111 1100 001' return Sys_DC;    // CVAP
        when '011 0111 1100 011' return Sys_DC;    // CGVAP
        when '011 0111 1100 101' return Sys_DC;    // CGDVAP
        when '011 0111 1101 001' return Sys_DC;    // CVADP
        when '011 0111 1101 011' return Sys_DC;    // CGVADP
        when '011 0111 1101 101' return Sys_DC;    // CGDVADP
        when '011 0111 1110 001' return Sys_DC;    // CIVAC
        when '011 0111 1110 011' return Sys_DC;    // CIGVAC
        when '011 0111 1110 101' return Sys_DC;    // CIGDVAC
        when '110 0111 1110 001' return Sys_DC;    // CIPAPA
        when '110 0111 1110 101' return Sys_DC;    // CIGDPAPA
        when '000 0111 0001 000' return Sys_IC;    // IALLUIS
        when '000 0111 0101 000' return Sys_IC;    // IALLU
        when '011 0111 0101 001' return Sys_IC;    // IVAU
        when '000 1000 0001 000' return Sys_TLBI;  // VMALLE1OS
        when '000 1000 0001 001' return Sys_TLBI;  // VAE1OS
        when '000 1000 0001 010' return Sys_TLBI;  // ASIDE1OS
        when '000 1000 0001 011' return Sys_TLBI;  // VAAE1OS
        when '000 1000 0001 101' return Sys_TLBI;  // VALE1OS
        when '000 1000 0001 111' return Sys_TLBI;  // VAALE1OS
        when '000 1000 0010 001' return Sys_TLBI;  // RVAE1IS
        when '000 1000 0010 011' return Sys_TLBI;  // RVAAE1IS
        when '000 1000 0010 101' return Sys_TLBI;  // RVALE1IS
        when '000 1000 0010 111' return Sys_TLBI;  // RVAALE1IS
        when '000 1000 0011 000' return Sys_TLBI;  // VMALLE1IS
        when '000 1000 0011 001' return Sys_TLBI;  // VAE1IS
        when '000 1000 0011 010' return Sys_TLBI;  // ASIDE1IS
        when '000 1000 0011 011' return Sys_TLBI;  // VAAE1IS
        when '000 1000 0011 101' return Sys_TLBI;  // VALE1IS
        when '000 1000 0011 111' return Sys_TLBI;  // VAALE1IS
```

```
        when '000 1000 0101 001' return Sys_TLBI;  // RVAE1OS
        when '000 1000 0101 011' return Sys_TLBI;  // RVAAE1OS
        when '000 1000 0101 101' return Sys_TLBI;  // RVALE1OS
        when '000 1000 0101 111' return Sys_TLBI;  // RVAALE1OS
        when '000 1000 0110 001' return Sys_TLBI;  // RVAE1
        when '000 1000 0110 011' return Sys_TLBI;  // RVAAE1
        when '000 1000 0110 101' return Sys_TLBI;  // RVALE1
        when '000 1000 0110 111' return Sys_TLBI;  // RVAALE1
        when '000 1000 0111 000' return Sys_TLBI;  // VMALLE1
        when '000 1000 0111 001' return Sys_TLBI;  // VAE1
        when '000 1000 0111 010' return Sys_TLBI;  // ASIDE1
        when '000 1000 0111 011' return Sys_TLBI;  // VAAE1
        when '000 1000 0111 101' return Sys_TLBI;  // VALE1
        when '000 1000 0111 111' return Sys_TLBI;  // VAALE1
        when '000 1001 0001 000' return Sys_TLBI;  // VMALLE1OSNXS
        when '000 1001 0001 001' return Sys_TLBI;  // VAE1OSNXS
        when '000 1001 0001 010' return Sys_TLBI;  // ASIDE1OSNXS
        when '000 1001 0001 011' return Sys_TLBI;  // VAAE1OSNXS
        when '000 1001 0001 101' return Sys_TLBI;  // VALE1OSNXS
        when '000 1001 0001 111' return Sys_TLBI;  // VAALE1OSNXS
        when '000 1001 0010 001' return Sys_TLBI;  // RVAE1ISNXS
        when '000 1001 0010 011' return Sys_TLBI;  // RVAAE1ISNXS
        when '000 1001 0010 101' return Sys_TLBI;  // RVALE1ISNXS
        when '000 1001 0010 111' return Sys_TLBI;  // RVAALE1ISNXS
        when '000 1001 0011 000' return Sys_TLBI;  // VMALLE1ISNXS
        when '000 1001 0011 001' return Sys_TLBI;  // VAE1ISNXS
        when '000 1001 0011 010' return Sys_TLBI;  // ASIDE1ISNXS
        when '000 1001 0011 011' return Sys_TLBI;  // VAAE1ISNXS
        when '000 1001 0011 101' return Sys_TLBI;  // VALE1ISNXS
        when '000 1001 0011 111' return Sys_TLBI;  // VAALE1ISNXS
        when '000 1001 0101 001' return Sys_TLBI;  // RVAE1OSNXS
        when '000 1001 0101 011' return Sys_TLBI;  // RVAAE1OSNXS
        when '000 1001 0101 101' return Sys_TLBI;  // RVALE1OSNXS
        when '000 1001 0101 111' return Sys_TLBI;  // RVAALE1OSNXS
        when '000 1001 0110 001' return Sys_TLBI;  // RVAE1NXS
        when '000 1001 0110 011' return Sys_TLBI;  // RVAAE1NXS
        when '000 1001 0110 101' return Sys_TLBI;  // RVALE1NXS
        when '000 1001 0110 111' return Sys_TLBI;  // RVAALE1NXS
        when '000 1001 0111 000' return Sys_TLBI;  // VMALLE1NXS
        when '000 1001 0111 001' return Sys_TLBI;  // VAE1NXS
        when '000 1001 0111 010' return Sys_TLBI;  // ASIDE1NXS
        when '000 1001 0111 011' return Sys_TLBI;  // VAAE1NXS
        when '000 1001 0111 101' return Sys_TLBI;  // VALE1NXS
        when '000 1001 0111 111' return Sys_TLBI;  // VAALE1NXS
        when '100 1000 0000 001' return Sys_TLBI;  // IPAS2E1IS
        when '100 1000 0000 010' return Sys_TLBI;  // RIPAS2E1IS
        when '100 1000 0000 101' return Sys_TLBI;  // IPAS2LE1IS
        when '100 1000 0000 110' return Sys_TLBI;  // RIPAS2LE1IS
        when '100 1000 0001 000' return Sys_TLBI;  // ALLE2OS
        when '100 1000 0001 001' return Sys_TLBI;  // VAE2OS
        when '100 1000 0001 100' return Sys_TLBI;  // ALLE1OS
        when '100 1000 0001 101' return Sys_TLBI;  // VALE2OS
        when '100 1000 0001 110' return Sys_TLBI;  // VMALLS12E1OS
        when '100 1000 0010 001' return Sys_TLBI;  // RVAE2IS
        when '100 1000 0010 101' return Sys_TLBI;  // RVALE2IS
        when '100 1000 0011 000' return Sys_TLBI;  // ALLE2IS
        when '100 1000 0011 001' return Sys_TLBI;  // VAE2IS
        when '100 1000 0011 100' return Sys_TLBI;  // ALLE1IS
        when '100 1000 0011 101' return Sys_TLBI;  // VALE2IS
        when '100 1000 0011 110' return Sys_TLBI;  // VMALLS12E1IS
        when '100 1000 0100 000' return Sys_TLBI;  // IPAS2E1OS
        when '100 1000 0100 001' return Sys_TLBI;  // IPAS2E1
        when '100 1000 0100 010' return Sys_TLBI;  // RIPAS2E1
        when '100 1000 0100 011' return Sys_TLBI;  // RIPAS2E1OS
        when '100 1000 0100 100' return Sys_TLBI;  // IPAS2LE1OS
        when '100 1000 0100 101' return Sys_TLBI;  // IPAS2LE1
        when '100 1000 0100 110' return Sys_TLBI;  // RIPAS2LE1
        when '100 1000 0100 111' return Sys_TLBI;  // RIPAS2LE1OS
        when '100 1000 0101 001' return Sys_TLBI;  // RVAE2OS
        when '100 1000 0101 101' return Sys_TLBI;  // RVALE2OS
```

```
        when '100 1000 0110 001' return Sys_TLBI;  // RVAE2
        when '100 1000 0110 101' return Sys_TLBI;  // RVALE2
        when '100 1000 0111 000' return Sys_TLBI;  // ALLE2
        when '100 1000 0111 001' return Sys_TLBI;  // VAE2
        when '100 1000 0111 100' return Sys_TLBI;  // ALLE1
        when '100 1000 0111 101' return Sys_TLBI;  // VALE2
        when '100 1000 0111 110' return Sys_TLBI;  // VMALLS12E1
        when '100 1001 0000 001' return Sys_TLBI;  // IPAS2E1ISNXS
        when '100 1001 0000 010' return Sys_TLBI;  // RIPAS2E1ISNXS
        when '100 1001 0000 101' return Sys_TLBI;  // IPAS2LE1ISNXS
        when '100 1001 0000 110' return Sys_TLBI;  // RIPAS2LE1ISNXS
        when '100 1001 0001 000' return Sys_TLBI;  // ALLE2OSNXS
        when '100 1001 0001 001' return Sys_TLBI;  // VAE2OSNXS
        when '100 1001 0001 100' return Sys_TLBI;  // ALLE1OSNXS
        when '100 1001 0001 101' return Sys_TLBI;  // VALE2OSNXS
        when '100 1001 0001 110' return Sys_TLBI;  // VMALLS12E1OSNXS
        when '100 1001 0010 001' return Sys_TLBI;  // RVAE2ISNXS
        when '100 1001 0010 101' return Sys_TLBI;  // RVALE2ISNXS
        when '100 1001 0011 000' return Sys_TLBI;  // ALLE2ISNXS
        when '100 1001 0011 001' return Sys_TLBI;  // VAE2ISNXS
        when '100 1001 0011 100' return Sys_TLBI;  // ALLE1ISNXS
        when '100 1001 0011 101' return Sys_TLBI;  // VALE2ISNXS
        when '100 1001 0011 110' return Sys_TLBI;  // VMALLS12E1ISNXS
        when '100 1001 0100 000' return Sys_TLBI;  // IPAS2E1OSNXS
        when '100 1001 0100 001' return Sys_TLBI;  // IPAS2E1NXS
        when '100 1001 0100 010' return Sys_TLBI;  // RIPAS2E1NXS
        when '100 1001 0100 011' return Sys_TLBI;  // RIPAS2E1OSNXS
        when '100 1001 0100 100' return Sys_TLBI;  // IPAS2LE1OSNXS
        when '100 1001 0100 101' return Sys_TLBI;  // IPAS2LE1NXS
        when '100 1001 0100 110' return Sys_TLBI;  // RIPAS2LE1NXS
        when '100 1001 0100 111' return Sys_TLBI;  // RIPAS2LE1OSNXS
        when '100 1001 0101 001' return Sys_TLBI;  // RVAE2OSNXS
        when '100 1001 0101 101' return Sys_TLBI;  // RVALE2OSNXS
        when '100 1001 0110 001' return Sys_TLBI;  // RVAE2NXS
        when '100 1001 0110 101' return Sys_TLBI;  // RVALE2NXS
        when '100 1001 0111 000' return Sys_TLBI;  // ALLE2NXS
        when '100 1001 0111 001' return Sys_TLBI;  // VAE2NXS
        when '100 1001 0111 100' return Sys_TLBI;  // ALLE1NXS
        when '100 1001 0111 101' return Sys_TLBI;  // VALE2NXS
        when '100 1001 0111 110' return Sys_TLBI;  // VMALLS12E1NXS
        when '110 1000 0001 000' return Sys_TLBI;  // ALLE3OS
        when '110 1000 0001 001' return Sys_TLBI;  // VAE3OS
        when '110 1000 0001 100' return Sys_TLBI;  // PAALLOS
        when '110 1000 0001 101' return Sys_TLBI;  // VALE3OS
        when '110 1000 0010 001' return Sys_TLBI;  // RVAE3IS
        when '110 1000 0010 101' return Sys_TLBI;  // RVALE3IS
        when '110 1000 0011 000' return Sys_TLBI;  // ALLE3IS
        when '110 1000 0011 001' return Sys_TLBI;  // VAE3IS
        when '110 1000 0011 101' return Sys_TLBI;  // VALE3IS
        when '110 1000 0100 011' return Sys_TLBI;  // RPAOS
        when '110 1000 0100 111' return Sys_TLBI;  // RPALOS
        when '110 1000 0101 001' return Sys_TLBI;  // RVAE3OS
        when '110 1000 0101 101' return Sys_TLBI;  // RVALE3OS
        when '110 1000 0110 001' return Sys_TLBI;  // RVAE3
        when '110 1000 0110 101' return Sys_TLBI;  // RVALE3
        when '110 1000 0111 000' return Sys_TLBI;  // ALLE3
        when '110 1000 0111 001' return Sys_TLBI;  // VAE3
        when '110 1000 0111 100' return Sys_TLBI;  // PAALL
        when '110 1000 0111 101' return Sys_TLBI;  // VALE3
        when '110 1001 0001 000' return Sys_TLBI;  // ALLE3OSNXS
        when '110 1001 0001 001' return Sys_TLBI;  // VAE3OSNXS
        when '110 1001 0001 101' return Sys_TLBI;  // VALE3OSNXS
        when '110 1001 0010 001' return Sys_TLBI;  // RVAE3ISNXS
        when '110 1001 0010 101' return Sys_TLBI;  // RVALE3ISNXS
        when '110 1001 0011 000' return Sys_TLBI;  // ALLE3ISNXS
        when '110 1001 0011 001' return Sys_TLBI;  // VAE3ISNXS
        when '110 1001 0011 101' return Sys_TLBI;  // VALE3ISNXS
        when '110 1001 0101 001' return Sys_TLBI;  // RVAE3OSNXS
        when '110 1001 0101 101' return Sys_TLBI;  // RVALE3OSNXS
        when '110 1001 0110 001' return Sys_TLBI;  // RVAE3NXS
```

```
        when '110 1001 0110 101' return Sys_TLBI;  // RVALE3NXS
        when '110 1001 0111 000' return Sys_TLBI;  // ALLE3NXS
        when '110 1001 0111 001' return Sys_TLBI;  // VAE3NXS
        when '110 1001 0111 101' return Sys_TLBI;  // VALE3NXS
        otherwise                return Sys_SYS;
```

Library pseudocode for aarch64/instrs/system/sysops/sysop/SystemOp

```
enumeration SystemOp {Sys_AT, Sys_BRB, Sys_DC, Sys_IC, Sys_TLBI, Sys_SYS};
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.DTLBI_ALL

```
// AArch32.DTLBI_ALL()
// ===================
// Invalidate all data TLB entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.

AArch32.DTLBI_ALL(SecurityState security, Regime regime, Shareability shareability, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_DALL;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.attr         = attr;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.DTLBI_ASID

```
// AArch32.DTLBI_ASID()
// ====================
// Invalidate all data TLB stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Rt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.DTLBI_ASID(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability,
                   TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_DASID;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = TLBILevel_Any;
    r.attr         = attr;
    r.asid         = Zeros(8) : Rt<7:0>;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.DTLBI_VA

```
// AArch32.DTLBI_VA()
// ==================
// Invalidate by VA all stage 1 data TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//     TLBILevel_Any  : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.DTLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
                 Shareability shareability, TLBILevel level,  TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_DVA;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.asid         = Zeros(8) : Rt<7:0>;
    r.address      = Zeros(32) : Rt<31:12> : Zeros(12);

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.ITLBI_ALL

```
// AArch32.ITLBI_ALL()
// ===================
// Invalidate all instruction TLB entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.

AArch32.ITLBI_ALL(SecurityState security, Regime regime, Shareability shareability, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_IALL;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.attr         = attr;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.ITLBI_ASID

```
// AArch32.ITLBI_ASID()
// ====================
// Invalidate all instruction TLB stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Rt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.ITLBI_ASID(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability,
                   TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_IASID;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = TLBILevel_Any;
    r.attr         = attr;
    r.asid         = Zeros(8) : Rt<7:0>;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.ITLBI_VA

```
// AArch32.ITLBI_VA()
// ==================
// Invalidate by VA all stage 1 instruction TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//     TLBILevel_Any  : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.ITLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
                 Shareability shareability, TLBILevel level,  TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_IVA;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.asid         = Zeros(8) : Rt<7:0>;
    r.address      = Zeros(32) : Rt<31:12> : Zeros(12);

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.TLBI_ALL

```
// AArch32.TLBI_ALL()
// ==================
// Invalidate all entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_ALL(SecurityState security, Regime regime, Shareability shareability, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    r.op           = TLBIOp_ALL;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.attr         = attr;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.TLBI_ASID

```
// AArch32.TLBI_ASID()
// ===================
// Invalidate all stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Rt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_ASID(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability,
                  TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_ASID;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = TLBILevel_Any;
    r.attr         = attr;
    r.asid         = Zeros(8) : Rt<7:0>;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.TLBI_IPAS2

```
// AArch32.TLBI_IPAS2()
// ====================
// Invalidate by IPA all stage 2 only TLB entries in the indicated shareability
// domain matching the indicated VMID in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// IPA and related parameters of the are derived from Rt.
// When the indicated level is
//     TLBILevel_Any  : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_IPAS2(SecurityState security, Regime regime, bits(16) vmid,
                   Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2};
    assert security == SS_NonSecure;

    TLBIRecord r;
    r.op           = TLBIOp_IPAS2;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.address      = Zeros(24) : Rt<27:0> : Zeros(12);
    r.ipaspace     = PAS_NonSecure;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.TLBI_VA

```
// AArch32.TLBI_VA()
// =================
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//     TLBILevel_Any  : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
                Shareability shareability, TLBILevel level,  TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_VA;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.asid         = Zeros(8) : Rt<7:0>;
    r.address      = Zeros(32) : Rt<31:12> : Zeros(12);

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.TLBI_VAA

```
// AArch32.TLBI_VAA()
// ==================
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID (where regime supports VMID) and all ASID in the indicated regime
// with the indicated security state.
// VA and related parameters are derived from Rt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//     TLBILevel_Any  : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_VAA(SecurityState security, Regime regime, bits(16) vmid,
                 Shareability shareability, TLBILevel level, TLBIMemAttr attr,  bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_VAA;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.address      = Zeros(32) : Rt<31:12> : Zeros(12);

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.TLBI_VMALL

```
// AArch32.TLBI_VMALL()
// ====================
// Invalidate all stage 1 entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability
// domain that match the indicated VMID (where applicable).
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// Note: stage 2 only entries are not in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_VMALL(SecurityState security, Regime regime, bits(16) vmid,
                   Shareability shareability, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_VMALL;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.vmid         = vmid;
    r.attr         = attr;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.TLBI_VMALLS12

```
// AArch32.TLBI_VMALLS12()
// =======================
// Invalidate all stage 1 and stage 2 entries for the indicated translation
// regime with the indicated security state for all TLBs within the indicated
// shareability domain that match the indicated VMID.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_VMALLS12(SecurityState security, Regime regime, bits(16) vmid,
                      Shareability shareability, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    r.op           = TLBIOp_VMALLS12;
    r.from_aarch64 = FALSE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.vmid         = vmid;
    r.attr         = attr;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_ALL

```
// AArch64.TLBI_ALL()
// ==================
// Invalidate all entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_ALL(SecurityState security, Regime regime, Shareability shareability, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    r.op           = TLBIOp_ALL;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.attr         = attr;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_ASID

```
// AArch64.TLBI_ASID()
// ===================
// Invalidate all stage 1 entries matching the indicated VMID (where regime supports)
// and ASID in the parameter Xt in the indicated translation regime with the
// indicated security state for all TLBs within the indicated shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_ASID(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability,
                  TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_ASID;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = TLBILevel_Any;
    r.attr         = attr;
    r.asid         = Xt<63:48>;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_IPAS2

```
// AArch64.TLBI_IPAS2()
// ====================
// Invalidate by IPA all stage 2 only TLB entries in the indicated shareability
// domain matching the indicated VMID in the indicated regime with the indicated security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// IPA and related parameters of the are derived from Xt.
// When the indicated level is
//     TLBILevel_Any  : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_IPAS2(SecurityState security, Regime regime, bits(16) vmid,
                   Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    r.op           = TLBIOp_IPAS2;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.address      = ZeroExtend(Xt<39:0> : Zeros(12), 64);

    case security of
        when SS_NonSecure
            r.ipaspace = PAS_NonSecure;
        when SS_Secure
            r.ipaspace = if Xt<63> == '1' then PAS_NonSecure else PAS_Secure;
        when SS_Realm
            r.ipaspace = PAS_Realm;
        otherwise
            // Root security state does not have stage 2 translation
            Unreachable();

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_PAALL

```
// AArch64.TLBI_PAALL()
// ====================
// TLB Invalidate ALL GPT Information.
// Invalidates cached copies of GPT entries from TLBs in the indicated
// Shareabilty domain.
// The invalidation applies to all TLB entries containing GPT information.

AArch64.TLBI_PAALL(Shareability shareability)
    assert HaveRME() && PSTATE.EL == EL3;

    TLBIRecord r;

    // r.security and r.regime do not apply for TLBI by PA operations
    r.op    = TLBIOp_PAALL;
    r.level = TLBILevel_Any;
    r.attr  = TLBI_AllAttr;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);

    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_RIPAS2

```
// AArch64.TLBI_RIPAS2()
// =====================
// Range invalidate by IPA all stage 2 only TLB entries in the indicated
// shareability domain matching the indicated VMID in the indicated regime with the indicated
// security state.
// Note: stage 1 and stage 2 combined entries are not in the scope of this operation.
// The range of IPA and related parameters of the are derived from Xt.
// When the indicated level is
//     TLBILevel_Any  : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_RIPAS2(SecurityState security, Regime regime, bits(16) vmid,
                    Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_RIPAS2;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;

    bits(2) tg        = Xt<47:46>;
    integer scale     = UInt(Xt<45:44>);
    integer num       = UInt(Xt<43:39>);
    integer baseaddr  = SInt(Xt<36:0>);

    boolean valid;

    (valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

    if !valid then return;

    case security of
        when SS_NonSecure
            r.ipaspace = PAS_NonSecure;
        when SS_Secure
            r.ipaspace = if Xt<63> == '1' then PAS_NonSecure else PAS_Secure;
        when SS_Realm
            r.ipaspace = PAS_Realm;
        otherwise
            // Root security state does not have stage 2 translation
            Unreachable();

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_RPA

```
// AArch64.TLBI_RPA()
// ==================
// TLB Range Invalidate GPT Information by PA.
// Invalidates cached copies of GPT entries from TLBs in the indicated
// Shareabilty domain.
// The invalidation applies to TLB entries containing GPT information relating
// to the indicated physical address range.
// When the indicated level is
//     TLBILevel_Any  : this applies to TLB entries containing GPT information
//                      from all levels of the GPT walk
//     TLBILevel_Last : this applies to TLB entries containing GPT information
//                      from the last level of the GPT walk

AArch64.TLBI_RPA(TLBILevel level, bits(64) Xt, Shareability shareability)
    assert HaveRME() && PSTATE.EL == EL3;

    TLBIRecord r;
    integer range_bits;
    integer p;

    // r.security and r.regime do not apply for TLBI by PA operations
    r.op    = TLBIOp_RPA;
    r.level = level;
    r.attr  = TLBI_AllAttr;

    // SIZE field
    case Xt<47:44> of
        when '0000' range_bits = 12; // 4KB
        when '0001' range_bits = 14; // 16KB
        when '0010' range_bits = 16; // 64KB
        when '0011' range_bits = 21; // 2MB
        when '0100' range_bits = 25; // 32MB
        when '0101' range_bits = 29; // 512MB
        when '0110' range_bits = 30; // 1GB
        when '0111' range_bits = 34; // 16GB
        when '1000' range_bits = 36; // 64GB
        when '1001' range_bits = 39; // 512GB
        otherwise   range_bits = 0;  // Reserved encoding

    // If SIZE selects a range smaller than PGS, then PGS is used instead
    case DecodePGS(GPCCR_EL3.PGS) of
        when PGS_4KB  p = 12;
        when PGS_16KB p = 14;
        when PGS_64KB p = 16;

    if range_bits < p then
        range_bits = p;

    bits(52) BaseADDR = Zeros(52);
    case GPCCR_EL3.PGS of
        when '00' BaseADDR<51:12> = Xt<39:0>;   // 4KB
        when '10' BaseADDR<51:14> = Xt<39:2>;   // 16KB
        when '01' BaseADDR<51:16> = Xt<39:4>;   // 64KB

    // The calculation here automatically aligns BaseADDR to the size of
    // the region specififed in SIZE. However, the architecture does not
    // require this alignment and if BaseADDR is not aligned to the region
    // specified by SIZE then no entries are required to be invalidated.
    bits(52) start_addr = BaseADDR AND NOT ZeroExtend(Ones(range_bits), 52);
    bits(52) end_addr   = start_addr + ZeroExtend(Ones(range_bits), 52);

    // PASpace is not considered in TLBI by PA operations
    r.address     = ZeroExtend(start_addr, 64);
    r.end_address = ZeroExtend(end_addr, 64);

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_RVA

```
// AArch64.TLBI_RVA()
// ==================
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// shareability domain matching the indicated VMID and ASID (where regime
// supports VMID, ASID) in the indicated regime with the indicated security state.
// ASID, and range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//     TLBILevel_Any  : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_RVA(SecurityState security, Regime regime, bits(16) vmid,
                 Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64)  Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_RVA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.asid         = Xt<63:48>;

    boolean valid;

    (valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

    if !valid then return;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_RVAA

```
// AArch64.TLBI_RVAA()
// ===================
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// shareability domain matching the indicated VMID (where regimesupports VMID)
// and all ASID in the indicated regime with the indicated security state.
// VA range related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//     TLBILevel_Any  : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_RVAA(SecurityState security, Regime  regime, bits(16) vmid,
                  Shareability shareability, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_RVAA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;

    bits(2) tg        = Xt<47:46>;
    integer scale     = UInt(Xt<45:44>);
    integer num       = UInt(Xt<43:39>);
    integer baseaddr  = SInt(Xt<36:0>);

    boolean valid;

    (valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

    if !valid then return;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_VA

```
// AArch64.TLBI_VA()
// =================
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID and ASID (where regime supports VMID, ASID) in the indicated regime
// with the indicated security state.
// ASID, VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//     TLBILevel_Any  : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
                Shareability shareability, TLBILevel level,  TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_VA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.asid         = Xt<63:48>;
    r.address      = ZeroExtend(Xt<43:0> : Zeros(12), 64);

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_VAA

```
// AArch64.TLBI_VAA()
// ==================
// Invalidate by VA all stage 1 TLB entries in the indicated shareability domain
// matching the indicated VMID (where regime supports VMID) and all ASID in the indicated regime
// with the indicated security state.
// VA and related parameters are derived from Xt.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// When the indicated level is
//     TLBILevel_Any  : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_VAA(SecurityState security, Regime regime, bits(16) vmid,
                 Shareability shareability, TLBILevel level, TLBIMemAttr attr,  bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_VAA;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.vmid         = vmid;
    r.level        = level;
    r.attr         = attr;
    r.address      = ZeroExtend(Xt<43:0> : Zeros(12), 64);

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_VMALL

```
// AArch64.TLBI_VMALL()
// ====================
// Invalidate all stage 1 entries for the indicated translation regime with the
// the indicated security state for all TLBs within the indicated shareability
// domain that match the indicated VMID (where applicable).
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// Note: stage 2 only entries are not in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_VMALL(SecurityState security, Regime regime, bits(16) vmid,
                   Shareability shareability, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op           = TLBIOp_VMALL;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.vmid         = vmid;
    r.attr         = attr;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_VMALLS12

```
// AArch64.TLBI_VMALLS12()
// =======================
// Invalidate all stage 1 and stage 2 entries for the indicated translation
// regime with the indicated security state for all TLBs within the indicated
// shareability domain that match the indicated VMID.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_VMALLS12(SecurityState security, Regime regime, bits(16) vmid,
                      Shareability shareability, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    r.op           = TLBIOp_VMALLS12;
    r.from_aarch64 = TRUE;
    r.security     = security;
    r.regime       = regime;
    r.level        = TLBILevel_Any;
    r.vmid         = vmid;
    r.attr         = attr;

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/ASID_NONE

```
constant bits(16) ASID_NONE = Zeros(16);
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/Broadcast

```
// Broadcast()
// ===========
// IMPLEMENTATION DEFINED function to broadcast TLBI operation within the indicated shareability
// domain.

Broadcast(Shareability shareability, TLBIRecord r)
    IMPLEMENTATION_DEFINED;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/DecodeTLBITG

```
// DecodeTLBITG()
// ==============
// Decode translation granule size in TLBI range instructions

TGx DecodeTLBITG(bits(2) tg)
    case tg of
        when '01'   return TGx_4KB;
        when '10'   return TGx_16KB;
        when '11'   return TGx_64KB;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/GPTTLBIMatch

```
// GPTTLBIMatch()
// ==============
// Determine whether the GPT TLB entry lies within the scope of inavlidation

boolean GPTTLBIMatch(TLBIRecord tlbi, GPTEntry entry)
    assert tlbi.op IN {TLBIOp_RPA, TLBIOp_PAALL};

    boolean match;
    case tlbi.op of
        when TLBIOp_RPA
            match = (UInt(tlbi.address<51:entry.size>)     <= UInt(entry.pa<51:entry.size>) &&
                     UInt(tlbi.end_address<51:entry.size>) >  UInt(entry.pa<51:entry.size>));
        when TLBIOp_PAALL
            match = TRUE;

    return match;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/HasLargeAddress

```
// HasLargeAddress()
// =================
// Returns TRUE if the regime is configured for 52 bit addresses, FALSE otherwise.

boolean HasLargeAddress(Regime regime)
    if !Have52BitIPAAndPASpaceExt() then
        return FALSE;
    case regime of
        when Regime_EL3
            return TCR_EL3<32> == '1';
        when Regime_EL2
            return TCR_EL2<32> == '1';
        when Regime_EL20
            return TCR_EL2<59> == '1';
        when Regime_EL10
            return TCR_EL1<59> == '1';
        otherwise
            Unreachable();
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBI

```
// TLBI()
// ======
// Performs TLB maintenance of operation on TLB to invalidate the matching transition table entries.

TLBI(TLBIRecord r)
    IMPLEMENTATION_DEFINED;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBILevel

```
enumeration TLBILevel {
    TLBILevel_Any,
    TLBILevel_Last
};
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBIMatch

```
// TLBIMatch()
// ===========
// Determine whether the TLB entry lies within the scope of inavlidation

boolean TLBIMatch(TLBIRecord tlbi, TLBRecord entry)
    boolean match;
    case tlbi.op of
        when TLBIOp_DALL, TLBIOp_IALL
            match = (tlbi.security == entry.context.ss &&
                     tlbi.regime   == entry.context.regime);
        when TLBIOp_DASID, TLBIOp_IASID
            match = (entry.context.includes_s1 &&
                     tlbi.security == entry.context.ss &&
                     tlbi.regime   == entry.context.regime &&
                     (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                     (UseASID(entry.context) && entry.context.nG == '1' &&
                        tlbi.asid  == entry.context.asid));
        when TLBIOp_DVA, TLBIOp_IVA
            match = (entry.context.includes_s1 &&
                     tlbi.security == entry.context.ss &&
                     tlbi.regime   == entry.context.regime &&
                     (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                     (!UseASID(entry.context) || tlbi.asid == entry.context.asid ||
                        entry.context.nG == '0') &&
                     tlbi.address<55:entry.blocksize> == entry.context.ia<55:entry.blocksize> &&
                     (tlbi.level == TLBILevel_Any || !entry.walkstate.istable));
        when TLBIOp_ALL
            relax_regime = (tlbi.from_aarch64 &&
                            tlbi.regime IN {Regime_EL20, Regime_EL2} &&
                            entry.context.regime IN {Regime_EL20, Regime_EL2});
            match = (tlbi.security == entry.context.ss &&
                     (tlbi.regime  == entry.context.regime || relax_regime));
        when TLBIOp_ASID
            match = (entry.context.includes_s1 &&
                     tlbi.security == entry.context.ss &&
                     tlbi.regime   == entry.context.regime &&
                     (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                     (UseASID(entry.context) && entry.context.nG == '1' &&
                        tlbi.asid  == entry.context.asid));
        when TLBIOp_IPAS2
            match = (!entry.context.includes_s1 && entry.context.includes_s2 &&
                     tlbi.security == entry.context.ss &&
                     tlbi.regime   == entry.context.regime &&
                     (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                     tlbi.ipaspace == entry.context.ipaspace &&
                     tlbi.address<51:entry.blocksize> == entry.context.ia<51:entry.blocksize> &&
                     (tlbi.level == TLBILevel_Any || !entry.walkstate.istable));
        when TLBIOp_VAA
            match = (entry.context.includes_s1 &&
                     tlbi.security == entry.context.ss &&
                     tlbi.regime   == entry.context.regime &&
                     (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                     tlbi.address<55:entry.blocksize> == entry.context.ia<55:entry.blocksize> &&
                     (tlbi.level == TLBILevel_Any || !entry.walkstate.istable));
        when TLBIOp_VA
            match = (entry.context.includes_s1 &&
                     tlbi.security == entry.context.ss &&
                     tlbi.regime   == entry.context.regime &&
                     (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                     (!UseASID(entry.context) || tlbi.asid == entry.context.asid ||
                        entry.context.nG == '0') &&
                     tlbi.address<55:entry.blocksize> == entry.context.ia<55:entry.blocksize> &&
                     (tlbi.level == TLBILevel_Any || !entry.walkstate.istable));
        when TLBIOp_VMALL
            match = (entry.context.includes_s1 &&
                     tlbi.security == entry.context.ss &&
                     tlbi.regime   == entry.context.regime &&
                     (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid));
        when TLBIOp_VMALLS12
            match = (tlbi.security == entry.context.ss &&
```

```
                    tlbi.regime   == entry.context.regime &&
                    (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid));
        when TLBIOp_RIPAS2
            match = (!entry.context.includes_s1 && entry.context.includes_s2 &&
                    tlbi.security == entry.context.ss &&
                    tlbi.regime   == entry.context.regime &&
                    (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                    tlbi.ipaspace == entry.context.ipaspace &&
                    (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == entry.context.tg) &&
                    UInt(tlbi.address)     <= UInt(entry.context.ia) &&
                    UInt(tlbi.end_address) >  UInt(entry.context.ia));
        when TLBIOp_RVAA
            match = (entry.context.includes_s1 &&
                    tlbi.security == entry.context.ss &&
                    tlbi.regime   == entry.context.regime &&
                    (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                    (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == entry.context.tg) &&
                    UInt(tlbi.address)     <= UInt(entry.context.ia) &&
                    UInt(tlbi.end_address) >  UInt(entry.context.ia));
        when TLBIOp_RVA
            match = (entry.context.includes_s1 &&
                    tlbi.security == entry.context.ss &&
                    tlbi.regime   == entry.context.regime &&
                    (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                    (!UseASID(entry.context) || tlbi.asid == entry.context.asid ||
                       entry.context.nG == '0') &&
                    (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == entry.context.tg) &&
                    UInt(tlbi.address)     <= UInt(entry.context.ia) &&
                    UInt(tlbi.end_address) >  UInt(entry.context.ia));
        when TLBIOp_RPA
            match = (entry.context.includes_gpt &&
                    UInt(tlbi.address)     <= UInt(entry.walkstate.baseaddress.address) &&
                    UInt(tlbi.end_address) >  UInt(entry.walkstate.baseaddress.address));
        when TLBIOp_PAALL
            match = entry.context.includes_gpt;

    if tlbi.attr == TLBI_ExcludeXS && entry.context.xs == '1' then
        match = FALSE;

    return match;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBIMemAttr

```
enumeration TLBIMemAttr {
    TLBI_AllAttr,
    TLBI_ExcludeXS
};
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBIOp

```
enumeration TLBIOp {
    TLBIOp_DALL,          // AArch32 Data TLBI operations - deprecated
    TLBIOp_DASID,
    TLBIOp_DVA,
    TLBIOp_IALL,          // AArch32 Instruction TLBI operations - deprecated
    TLBIOp_IASID,
    TLBIOp_IVA,
    TLBIOp_ALL,
    TLBIOp_ASID,
    TLBIOp_IPAS2,
    TLBIOp_VAA,
    TLBIOp_VA,
    TLBIOp_VMALL,
    TLBIOp_VMALLS12,
    TLBIOp_RIPAS2,
    TLBIOp_RVAA,
    TLBIOp_RVA,
    TLBIOp_RPA,
    TLBIOp_PAALL,
};
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBIRange

```
// TLBIRange()
// ===========
// Extract the input address range information from encoded Xt.

(boolean, bits(2), bits(64), bits(64)) TLBIRange(Regime regime, bits(64) Xt)
    boolean  valid = TRUE;
    bits(64) start = Zeros(64);
    bits(64) end   = Zeros(64);

    bits(2) tg        = Xt<47:46>;
    integer scale     = UInt(Xt<45:44>);
    integer num       = UInt(Xt<43:39>);
    integer tg_bits;

    if tg == '00' then
        return (FALSE, tg, start, end);

    case tg of
        when '01' // 4KB
            tg_bits = 12;
            if HasLargeAddress(regime) then
                start<52:16> = Xt<36:0>;
                start<63:53> = Replicate(Xt<36>, 11);
            else
                start<48:12> = Xt<36:0>;
                start<63:49> = Replicate(Xt<36>, 15);
        when '10' // 16KB
            tg_bits = 14;
            if HasLargeAddress(regime) then
                start<52:16> = Xt<36:0>;
                start<63:53> = Replicate(Xt<36>, 11);
            else
                start<50:14> = Xt<36:0>;
                start<63:51> = Replicate(Xt<36>, 13);
        when '11' // 64KB
            tg_bits = 16;
            start<52:16> = Xt<36:0>;
            start<63:53> = Replicate(Xt<36>, 11);
        otherwise
            Unreachable();

    integer range = (num+1) << (5*scale + 1 + tg_bits);
    end   = start + range<63:0>;

    if end<52> != start<52> then
        // overflow, saturate it
        end = Replicate(start<52>, 64-52) : Ones(52);

    return (valid, tg, start, end);
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBIRecord

```
type TLBIRecord is (
    TLBIOp          op,
    boolean         from_aarch64, // originated as an AArch64 operation
    SecurityState   security,
    Regime          regime,
    bits(16)        vmid,
    bits(16)        asid,
    TLBILevel       level,
    TLBIMemAttr     attr,
    PASpace         ipaspace,     // For operations that take IPA as input address
    bits(64)        address,      // input address, for range operations, start address
    bits(64)        end_address,  // for range operations, end address
    bits(2)         tg,           // for range operations, translation granule
)
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/VMID

```
// VMID[]
// ======
// Effective VMID.

bits(16) VMID[]
    if EL2Enabled() then
        if !ELUsingAArch32(EL2) then
            if Have16bitVMID() && VTCR_EL2.VS == '1' then
                return VTTBR_EL2.VMID;
            else
                return ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
        else
            return ZeroExtend(VTTBR.VMID, 16);
    elsif HaveEL(EL2) && HaveSecureEL2Ext() then
        return Zeros(16);
    else
        return VMID_NONE;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/VMID_NONE

```
constant bits(16) VMID_NONE = Zeros(16);
```

Library pseudocode for aarch64/instrs/vector/arithmetic/binary/uniform/logical/bsl-eor/vbitop/VBitOp

```
enumeration VBitOp      {VBitOp_VBIF, VBitOp_VBIT, VBitOp_VBSL, VBitOp_VEOR};
```

Library pseudocode for aarch64/instrs/vector/arithmetic/unary/cmp/compareop/CompareOp

```
enumeration CompareOp   {CompareOp_GT, CompareOp_GE, CompareOp_EQ,
                         CompareOp_LE, CompareOp_LT};
```

Library pseudocode for aarch64/instrs/vector/logical/immediateop/ImmediateOp

```
enumeration ImmediateOp {ImmediateOp_MOVI, ImmediateOp_MVNI,
                         ImmediateOp_ORR, ImmediateOp_BIC};
```

Library pseudocode for aarch64/instrs/vector/reduce/reduceop/Reduce

```
// Reduce()
// ========

bits(esize) Reduce(ReduceOp op, bits(N) input, integer esize)
    boolean altfp = HaveAltFP() && !UsingAArch32() && FPCR.AH == '1';
    return Reduce(op, input, esize, altfp);

// Reduce()
// ========
// Perform the operation 'op' on pairs of elements from the input vector,
// reducing the vector to a scalar result. The 'altfp' argument controls
// alternative floating-point behaviour.

bits(esize) Reduce(ReduceOp op, bits(N) input, integer esize, boolean altfp)
    integer half;
    bits(esize) hi;
    bits(esize) lo;
    bits(esize) result;

    if N == esize then
        return input<esize-1:0>;

    half = N DIV 2;
    hi = Reduce(op, input<N-1:half>, esize, altfp);
    lo = Reduce(op, input<half-1:0>, esize, altfp);

    case op of
        when ReduceOp_FMINNUM
            result = FPMinNum(lo, hi, FPCR[]);
        when ReduceOp_FMAXNUM
            result = FPMaxNum(lo, hi, FPCR[]);
        when ReduceOp_FMIN
            result = FPMin(lo, hi, FPCR[], altfp);
        when ReduceOp_FMAX
            result = FPMax(lo, hi, FPCR[], altfp);
        when ReduceOp_FADD
            result = FPAdd(lo, hi, FPCR[]);
        when ReduceOp_ADD
            result = lo + hi;

    return result;
```

Library pseudocode for aarch64/instrs/vector/reduce/reduceop/ReduceOp

```
enumeration ReduceOp {ReduceOp_FMINNUM, ReduceOp_FMAXNUM,
                      ReduceOp_FMIN, ReduceOp_FMAX,
                      ReduceOp_FADD, ReduceOp_ADD};
```

Library pseudocode for aarch64/translation/debug/AArch64.CheckBreakpoint

```
// AArch64.CheckBreakpoint()
// =========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch64
// translation regime, when either debug exceptions are enabled, or halting debug is enabled
// and halting is allowed.

FaultRecord AArch64.CheckBreakpoint(bits(64) vaddress, AccType acctype_in, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert (UsingAArch32() && size IN {2,4}) || size == 4;
    AccType acctype = acctype_in;

    match = FALSE;

    for i = 0 to NumBreakpointsImplemented() - 1
        match_i = AArch64.BreakpointMatch(i, vaddress, acctype, size);
        match = match || match_i;

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Breakpoint;
        Halt(reason);
    elsif match then
        acctype = AccType_IFETCH;
        iswrite = FALSE;
        return AArch64.DebugFault(acctype, iswrite);
    else
        return NoFault();
```

Library pseudocode for aarch64/translation/debug/AArch64.CheckDebug

```
// AArch64.CheckDebug()
// ====================
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch64.CheckDebug(bits(64) vaddress, AccType acctype, boolean iswrite, integer size)

    FaultRecord fault = NoFault();
    boolean generate_exception;

    boolean d_side = (IsDataAccess(acctype) || acctype == AccType_DC);
    boolean i_side = (acctype == AccType_IFETCH);
    if HaveNV2Ext() && acctype == AccType_NV2REGISTER then
        mask = '0';
        ss = CurrentSecurityState();
        generate_exception = AArch64.GenerateDebugExceptionsFrom(EL2, ss, mask) && MDSCR_EL1.MDE == '1';
    else
        generate_exception = AArch64.GenerateDebugExceptions() && MDSCR_EL1.MDE == '1';
    halt = HaltOnBreakpointOrWatchpoint();

    if generate_exception || halt then
        if d_side then
            fault = AArch64.CheckWatchpoint(vaddress, acctype, iswrite, size);
        elsif i_side then
            fault = AArch64.CheckBreakpoint(vaddress, acctype, size);

    return fault;
```

Library pseudocode for aarch64/translation/debug/AArch64.CheckWatchpoint

```
// AArch64.CheckWatchpoint()
// =========================
// Called before accessing the memory location of "size" bytes at "address",
// when either debug exceptions are enabled for the access, or halting debug
// is enabled and halting is allowed.

FaultRecord AArch64.CheckWatchpoint(bits(64) vaddress, AccType acctype,
                                    boolean iswrite_in, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    boolean iswrite = iswrite_in;

    if acctype == AccType_DC then
        if !iswrite then
            return NoFault();
    elsif !IsDataAccess(acctype) then
        return NoFault();

    match = FALSE;
    match_on_read = FALSE;
    ispriv = AArch64.AccessUsesEL(acctype) != EL0;

    for i = 0 to NumWatchpointsImplemented() - 1
        if AArch64.WatchpointMatch(i, vaddress, size, ispriv, acctype, iswrite) then
            match = TRUE;
            if DBGWCR_EL1[i].LSC<0> == '1' then
                match_on_read = TRUE;

    if match && IsAtomicRW(acctype) then
        iswrite = !match_on_read;

    if match && HaltOnBreakpointOrWatchpoint() then
        if acctype != AccType_NONFAULT && acctype != AccType_CNOTFIRST then
            reason = DebugHalt_Watchpoint;
            EDWAR = vaddress;
            Halt(reason);
        else
            // Fault will be reported and cancelled
            return AArch64.DebugFault(acctype, iswrite);
    elsif match then
        return AArch64.DebugFault(acctype, iswrite);
    else
        return NoFault();
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.BlockBase

```
// AArch64.BlockBase()
// ===================
// Extract the address embedded in a block descriptor pointing to the base of
// a memory block

bits(52) AArch64.BlockBase(bits(64) descriptor, bit ds, TGx tgx, integer level)
    bits(52) blockbase = Zeros(52);

    if tgx == TGx_4KB && level == 2 then
        blockbase<47:21> = descriptor<47:21>;
    elsif tgx == TGx_4KB && level == 1 then
        blockbase<47:30> = descriptor<47:30>;
    elsif tgx == TGx_4KB && level == 0 then
        blockbase<47:39> = descriptor<47:39>;
    elsif tgx == TGx_16KB && level == 2 then
        blockbase<47:25> = descriptor<47:25>;
    elsif tgx == TGx_16KB && level == 1 then
        blockbase<47:36> = descriptor<47:36>;
    elsif tgx == TGx_64KB && level == 2 then
        blockbase<47:29> = descriptor<47:29>;
    elsif tgx == TGx_64KB && level == 1 then
        blockbase<47:42> = descriptor<47:42>;
    else
        Unreachable();

    if Have52BitPAExt() && tgx == TGx_64KB then
        blockbase<51:48> = descriptor<15:12>;
    elsif ds == '1' then
        blockbase<51:48> = descriptor<9:8>:descriptor<49:48>;

    return blockbase;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.IASize

```
// AArch64.IASize()
// ================
// Retrieve the number of bits containing the input address

integer AArch64.IASize(bits(6) txsz)
    return 64 - UInt(txsz);
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.NextTableBase

```
// AArch64.NextTableBase()
// =======================
// Extract the address embedded in a table descriptor pointing to the base of
// the next level table of descriptors

bits(52) AArch64.NextTableBase(bits(64) descriptor, bit ds, TGx tgx)
    bits(52) tablebase = Zeros(52);

    case tgx of
        when TGx_4KB  tablebase<47:12> = descriptor<47:12>;
        when TGx_16KB tablebase<47:14> = descriptor<47:14>;
        when TGx_64KB tablebase<47:16> = descriptor<47:16>;

    if Have52BitPAExt() && tgx == TGx_64KB then
        tablebase<51:48> = descriptor<15:12>;
    elsif ds == '1' then
        tablebase<51:48> = descriptor<9:8>:descriptor<49:48>;

    return tablebase;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.PageBase

```
// AArch64.PageBase()
// ==================
// Extract the address embedded in a page descriptor pointing to the base of
// a memory page

bits(52) AArch64.PageBase(bits(64) descriptor, bit ds, TGx tgx)
    bits(52) pagebase = Zeros(52);

    case tgx of
        when TGx_4KB  pagebase<47:12> = descriptor<47:12>;
        when TGx_16KB pagebase<47:14> = descriptor<47:14>;
        when TGx_64KB pagebase<47:16> = descriptor<47:16>;

    if Have52BitPAExt() && tgx == TGx_64KB then
        pagebase<51:48> = descriptor<15:12>;
    elsif ds == '1' then
        pagebase<51:48> = descriptor<9:8>:descriptor<49:48>;

    return pagebase;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.PhysicalAddressSize

```
// AArch64.PhysicalAddressSize()
// =============================
// Retrieve the number of bits bounding the physical address

integer AArch64.PhysicalAddressSize(bits(3) encoded_ps, TGx tgx)
    integer ps;
    integer max_ps;

    case encoded_ps of
        when '000'  ps = 32;
        when '001'  ps = 36;
        when '010'  ps = 40;
        when '011'  ps = 42;
        when '100'  ps = 44;
        when '101'  ps = 48;
        when '110'  ps = 52;
        otherwise
            ps = integer IMPLEMENTATION_DEFINED "Reserved Intermediate Physical Address size value";

    if tgx != TGx_64KB && !Have52BitIPAAndPASpaceExt() then
        max_ps = Min(48, AArch64.PAMax());
    else
        max_ps = AArch64.PAMax();

    return Min(ps, max_ps);
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.S1StartLevel

```
// AArch64.S1StartLevel()
// ======================
// Compute the initial lookup level when performing a stage 1 translation
// table walk

integer AArch64.S1StartLevel(S1TTWParams walkparams)
    // Input Address size
    iasize   = AArch64.IASize(walkparams.txsz);
    granulebits = TGxGranuleBits(walkparams.tgx);
    stride      = granulebits - 3;

    return FINAL_LEVEL - (((iasize-1) - granulebits) DIV stride);
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.S2SLTTEntryAddress

```
// AArch64.S2SLTTEntryAddress()
// ============================
// Compute the first stage 2 translation table descriptor address within the
// table pointed to by the base at the start level

FullAddress AArch64.S2SLTTEntryAddress(S2TTWParams walkparams, bits(52) ipa,
                                       FullAddress tablebase)
    startlevel  = AArch64.S2StartLevel(walkparams);
    iasize      = AArch64.IASize(walkparams.txsz);
    granulebits = TGxGranuleBits(walkparams.tgx);
    stride      = granulebits - 3;
    levels      = FINAL_LEVEL - startlevel;

    bits(52) index;
    lsb   = levels*stride + granulebits;
    msb   = iasize - 1;
    index = ZeroExtend(ipa<msb:lsb>:Zeros(3), 52);

    FullAddress descaddress;
    descaddress.address = tablebase.address OR index;
    descaddress.paspace = tablebase.paspace;

    return descaddress;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.S2StartLevel

```
// AArch64.S2StartLevel()
// ======================
// Determine the initial lookup level when performing a stage 2 translation
// table walk

integer AArch64.S2StartLevel(S2TTWParams walkparams)
    case walkparams.tgx of
        when TGx_4KB
            case walkparams.sl2:walkparams.sl0 of
                when '000' return 2;
                when '001' return 1;
                when '010' return 0;
                when '011' return 3;
                when '100' return -1;
        when TGx_16KB
            case walkparams.sl0 of
                when '00' return 3;
                when '01' return 2;
                when '10' return 1;
                when '11' return 0;
        when TGx_64KB
            case walkparams.sl0 of
                when '00' return 3;
                when '01' return 2;
                when '10' return 1;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.TTBaseAddress

```
// AArch64.TTBaseAddress()
// =======================
// Retrieve the PA/IPA pointing to the base of the initial translation table

bits(52) AArch64.TTBaseAddress(bits(64) ttb, bits(6) txsz, bits(3) ps,
                               bit ds, TGx tgx, integer startlevel)
    bits(52) tablebase = Zeros(52);

    // Input Address size
    iasize      = AArch64.IASize(txsz);
    granulebits = TGxGranuleBits(tgx);
    stride      = granulebits - 3;
    levels      = FINAL_LEVEL - startlevel;

    // Base address is aligned to size of the initial translation table in bytes
    tsize = (iasize - (levels*stride + granulebits)) + 3;

    if (Have52BitPAExt() && tgx == TGx_64KB && ps == '110') || (ds == '1') then
        tsize = Max(tsize, 6);
        tablebase<51:6> = ttb<5:2>:ttb<47:6>;
    else
        tablebase<47:1> = ttb<47:1>;

    tablebase = Align(tablebase, 1 << tsize);
    return tablebase;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.TTEntryAddress

```
// AArch64.TTEntryAddress()
// ========================
// Compute translation table descriptor address within the table pointed to by
// the table base

FullAddress AArch64.TTEntryAddress(integer level, TGx tgx, bits(6) txsz,
                                   bits(64) ia, FullAddress tablebase)
    // Input Address size
    iasize      = AArch64.IASize(txsz);
    granulebits = TGxGranuleBits(tgx);
    stride      = granulebits - 3;
    levels      = FINAL_LEVEL - level;

    bits(52) index;
    lsb   = levels*stride + granulebits;
    msb   = Min(iasize - 1, (lsb + stride) - 1);
    index = ZeroExtend(ia<msb:lsb>:Zeros(3), 52);

    FullAddress descaddress;
    descaddress.address = tablebase.address OR index;
    descaddress.paspace = tablebase.paspace;

    return descaddress;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.AddrTop

```
// AArch64.AddrTop()
// =================
// Get the top bit position of the virtual address.
// Bits above are not accounted as part of the translation process.

integer AArch64.AddrTop(bit tbid, AccType acctype, bit tbi)
    if tbid == '1' && acctype == AccType_IFETCH then
        return 63;

    if tbi == '1' then
        return 55;
    else
        return 63;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.ContiguousBitFaults

```
// AArch64.ContiguousBitFaults()
// =============================
// If contiguous bit is set, returns whether the translation size exceeds the
// input address size and if the implementation generates a fault

boolean AArch64.ContiguousBitFaults(bits(6) txsz, TGx tgx, integer level)
    // Input Address size
    iasize = AArch64.IASize(txsz);
    // Translation size
    tsize  = TranslationSize(tgx, level) + ContiguousSize(tgx, level);

    fault = boolean IMPLEMENTATION_DEFINED "Translation fault on misprogrammed contiguous bit";

    return tsize > iasize && fault;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.DebugFault

```
// AArch64.DebugFault()
// ====================
// Return a fault record indicating a hardware watchpoint/breakpoint

FaultRecord AArch64.DebugFault(AccType acctype, boolean iswrite)
    FaultRecord fault;

    fault.statuscode  = Fault_Debug;
    fault.acctype     = acctype;
    fault.write       = iswrite;
    fault.secondstage = FALSE;
    fault.s2fs1walk   = FALSE;

    return fault;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.ExclusiveFault

```
// AArch64.ExclusiveFault()
// ========================

FaultRecord AArch64.ExclusiveFault(AccType acctype, boolean iswrite,
                                   boolean secondstage, boolean s2fs1walk)
    FaultRecord fault;

    fault.statuscode  = Fault_Exclusive;
    fault.acctype     = acctype;
    fault.write       = iswrite;
    fault.secondstage = secondstage;
    fault.s2fs1walk   = s2fs1walk;

    return fault;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.IPAIsOutOfRange

```
// AArch64.IPAIsOutOfRange()
// =========================
// Check bits not resolved by translation are ZERO

boolean AArch64.IPAIsOutOfRange(bits(52) ipa, S2TTWParams walkparams)
    //Input Address size
    iasize = AArch64.IASize(walkparams.txsz);

    if iasize < 52 then
        return !IsZero(ipa<51:iasize>);
    else
        return FALSE;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.OAOutOfRange

```
// AArch64.OAOutOfRange()
// ======================
// Returns whether output address is expressed in the configured size number of bits

boolean AArch64.OAOutOfRange(TTWState walkstate, bits(3) ps, TGx tgx, bits(64) ia)
    // Output Address size
    oasize = AArch64.PhysicalAddressSize(ps, tgx);

    if oasize < 52 then
        if walkstate.istable then
            baseaddress = walkstate.baseaddress.address;
            return !IsZero(baseaddress<51:oasize>);
        else
            // Output address
            oa = StageOA(ia, tgx, walkstate);
            return !IsZero(oa.address<51:oasize>);
    else
        return FALSE;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S1HasAlignmentFault

```
// AArch64.S1HasAlignmentFault()
// =============================
// Returns whether stage 1 output fails alignment requirement on data accesses
// to Device memory

boolean AArch64.S1HasAlignmentFault(AccType acctype, boolean aligned,
                                    bit ntlsmd, MemoryAttributes memattrs)
    if acctype == AccType_IFETCH || memattrs.memtype != MemType_Device then
        return FALSE;

    if acctype == AccType_A32LSMD && ntlsmd == '0' && memattrs.device != DeviceType_GRE then
        return TRUE;

    return !aligned || acctype == AccType_DCZVA;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S1HasPermissionsFault

```
// AArch64.S1HasPermissionsFault()
// ===============================
// Returns whether stage 1 access violates permissions of target memory

boolean AArch64.S1HasPermissionsFault(Regime regime, SecurityState ss, TTWState walkstate,
                                      S1TTWParams walkparams, boolean ispriv, AccType acctype,
                                      boolean iswrite)
    bit r;
    bit w;
    bit x;
    permissions = walkstate.permissions;

    if HasUnprivileged(regime) then
        bit pr;
        bit pw;
        bit ur;
        bit uw;
        // Apply leaf permissions
        case permissions.ap<2:1> of
            when '00' (pr,pw,ur,uw) = ('1','1','0','0'); // Privileged access
            when '01' (pr,pw,ur,uw) = ('1','1','1','1'); // No effect
            when '10' (pr,pw,ur,uw) = ('1','0','0','0'); // Read-only, privileged access
            when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // Read-only

        // Apply hierarchical permissions
        case permissions.ap_table of
            when '00' (pr,pw,ur,uw) = ( pr, pw, ur, uw); // No effect
            when '01' (pr,pw,ur,uw) = ( pr, pw,'0','0'); // Privileged access
            when '10' (pr,pw,ur,uw) = ( pr,'0', ur,'0'); // Read-only
            when '11' (pr,pw,ur,uw) = ( pr,'0','0','0'); // Read-only, privileged access

        // Locations writable by unprivileged cannot be executed by privileged
        px = NOT(permissions.pxn OR permissions.pxn_table OR uw);
        ux = NOT(permissions.uxn OR permissions.uxn_table);

        pan_access = !(acctype IN {AccType_DC, AccType_IFETCH, AccType_AT, AccType_NV2REGISTER});
        if HavePANExt() && pan_access && !(regime == Regime_EL10 && walkparams.nv1 == '1') then
            bit pan;
            if (boolean IMPLEMENTATION_DEFINED "SCR_EL3.SIF affects EPAN" &&
                    CurrentSecurityState() == SS_Secure &&
                    walkstate.baseaddress.paspace == PAS_NonSecure &&
                    walkparams.sif == '1') then
                ux = '0';

            if (boolean IMPLEMENTATION_DEFINED "Realm EL2&0 regime affects EPAN" &&
                    CurrentSecurityState() == SS_Realm && regime == Regime_EL20 &&
                    walkstate.baseaddress.paspace != PAS_Realm) then
                ux = '0';
            pan = PSTATE.PAN AND (ur OR uw OR (walkparams.epan AND ux));
            pr = pr AND NOT(pan);
            pw = pw AND NOT(pan);

        (r,w,x) = if ispriv then (pr,pw,px) else (ur,uw,ux);
    else
        // Apply leaf permissions
        case permissions.ap<2> of
            when '0' (r,w) = ('1','1'); // No effect
            when '1' (r,w) = ('1','0'); // Read-only

        // Apply hierarchical permissions
        case permissions.ap_table<1> of
            when '0' (r,w) = ( r , w ); // No effect
            when '1' (r,w) = ( r ,'0'); // Read-only

        x = NOT(permissions.xn OR permissions.xn_table);

    // Prevent execution from writable locations if WXN is set
    x = x AND NOT(walkparams.wxn AND w);
    // Prevent execution from Non-secure space by PE in secure state if SIF is set
    if ss == SS_Secure && walkstate.baseaddress.paspace == PAS_NonSecure then
```

```
        x = x AND NOT(walkparams.sif);
    // Prevent execution from non-Root space by Root
    if ss == SS_Root && walkstate.baseaddress.paspace != PAS_Root then
        x = '0';
    // Prevent execution from non-Realm space by Realm EL2 and Realm EL2&0
    if (ss == SS_Realm && regime IN {Regime_EL2, Regime_EL20} &&
            walkstate.baseaddress.paspace != PAS_Realm) then
        x = '0';

    if acctype == AccType_IFETCH then
        if (ConstrainUnpredictable(Unpredictable_INSTRDEVICE) == Constraint_FAULT &&
                walkstate.memattrs.memtype == MemType_Device) then
            return TRUE;

        return x == '0';
    elsif acctype == AccType_DC then
        if iswrite then
            return w == '0';
        else
            // DC from privileged context which do no write cannot Permission fault
            return !ispriv && (r == '0' ||
                (IsCMOWControlledInstruction() && walkparams.cmow == '1' && w == '0'));
    elsif acctype == AccType_IC then
        // IC instructions do not write
        assert !iswrite;
        impdef_ic_fault = boolean IMPLEMENTATION_DEFINED "Permission fault on EL0 IC_IVAU execution";

        // IC from privileged context cannot Permission fault
        return !ispriv && ((r == '0' && impdef_ic_fault) ||
            (IsCMOWControlledInstruction() && walkparams.cmow == '1' && w == '0'));
    elsif iswrite then
        return w == '0';
    else
        return r == '0';
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S1InvalidTxSZ

```
// AArch64.S1InvalidTxSZ()
// =======================
// Detect erroneous configuration of stage 1 TxSZ field if the implementation
// does not constrain the value of TxSZ

boolean AArch64.S1InvalidTxSZ(S1TTWParams walkparams)
    mintxsz = AArch64.S1MinTxSZ(walkparams.ds, walkparams.tgx);
    maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);

    return UInt(walkparams.txsz) < mintxsz || UInt(walkparams.txsz) > maxtxsz;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S2HasAlignmentFault

```
// AArch64.S2HasAlignmentFault()
// =============================
// Returns whether stage 2 output fails alignment requirement on data accesses
// to Device memory

boolean AArch64.S2HasAlignmentFault(AccType acctype, boolean aligned, MemoryAttributes memattrs)
    if acctype == AccType_IFETCH || memattrs.memtype != MemType_Device then
        return FALSE;

    return !aligned || acctype == AccType_DCZVA;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S2HasPermissionsFault

```
// AArch64.S2HasPermissionsFault()
// ===============================
// Returns whether stage 2 access violates permissions of target memory

boolean AArch64.S2HasPermissionsFault(boolean s2fs1walk, TTWState walkstate, SecurityState ss,
                                      S2TTWParams walkparams, boolean ispriv, AccType acctype,
                                      boolean iswrite)
    permissions = walkstate.permissions;
    memtype = walkstate.memattrs.memtype;

    r = permissions.s2ap<0>;
    w = permissions.s2ap<1>;

    bit px;
    bit ux;
    case (permissions.s2xn:permissions.s2xnx) of
        when '00' (px,ux) = ('1','1');
        when '01' (px,ux) = ('0','1');
        when '10' (px,ux) = ('0','0');
        when '11' (px,ux) = ('1','0');

    x = if ispriv then px else ux;

    if s2fs1walk && walkparams.ptw == '1' && memtype == MemType_Device then
        return TRUE;
    // Prevent translation table walks in Non-secure space by Realm state
    elsif s2fs1walk && ss == SS_Realm && walkstate.baseaddress.paspace != PAS_Realm then
        return TRUE;
    elsif acctype == AccType_IFETCH then
        constraint = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
        if constraint == Constraint_FAULT && memtype == MemType_Device then
            return TRUE;
        // Prevent execution from Non-secure space by Realm state
        if ss == SS_Realm && walkstate.baseaddress.paspace != PAS_Realm then
            return TRUE;
        return x == '0';
    elsif acctype == AccType_DC then
        // AArch32 DC maintenance instructions operating by VA cannot fault.
        if iswrite then
            return !ELUsingAArch32(EL1) && w == '0';
        else
            return ((!ispriv && !ELUsingAArch32(EL1) && r == '0') ||
                (IsCMOWControlledInstruction() && walkparams.cmow == '1' && w == '0'));
    elsif acctype == AccType_IC then
        // IC instructions do not write
        assert !iswrite;
        impdef_ic_fault = boolean IMPLEMENTATION_DEFINED "Permission fault on EL0 IC_IVAU execution";

        return ((!ispriv && !ELUsingAArch32(EL1) && r == '0' && impdef_ic_fault) ||
            (IsCMOWControlledInstruction() && walkparams.cmow == '1' && w == '0'));
    elsif iswrite then
        return w == '0';
    else
        return r == '0';
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S2InconsistentSL

```
// AArch64.S2InconsistentSL()
// ==========================
// Detect inconsistent configuration of stage 2 TxSZ and SL fields

boolean AArch64.S2InconsistentSL(S2TTWParams walkparams)
    startlevel  = AArch64.S2StartLevel(walkparams);
    levels      = FINAL_LEVEL - startlevel;
    granulebits = TGxGranuleBits(walkparams.tgx);
    stride      = granulebits - 3;

    // Input address size must at least be large enough to be resolved from the start level
    sl_min_iasize = (
        levels * stride // Bits resolved by table walk, except initial level
        + granulebits   // Bits directly mapped to output address
        + 1);           // At least 1 more bit to be decoded by initial level

    // Can accomodate 1 more stride in the level + concatenation of up to 2^4 tables
    sl_max_iasize = sl_min_iasize + (stride-1) + 4;
    // Configured Input Address size
    iasize        = AArch64.IASize(walkparams.txsz);

    return iasize < sl_min_iasize || iasize > sl_max_iasize;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S2InvalidSL

```
// AArch64.S2InvalidSL()
// =====================
// Detect invalid configuration of SL field

boolean AArch64.S2InvalidSL(S2TTWParams walkparams)
    case walkparams.tgx of
        when TGx_4KB
            case walkparams.sl2:walkparams.sl0 of
                when '1x1' return TRUE;
                when '11x' return TRUE;
                when '010' return AArch64.PAMax() < 44;
                when '011' return !HaveSmallTranslationTableExt();
                otherwise  return FALSE;
        when TGx_16KB
            case walkparams.sl0 of
                when '11' return walkparams.ds == '0';
                when '10' return AArch64.PAMax() < 42;
                otherwise  return FALSE;
        when TGx_64KB
            case walkparams.sl0 of
                when '11'  return TRUE;
                when '10'  return AArch64.PAMax() < 44;
                otherwise  return FALSE;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S2InvalidTxSZ

```
// AArch64.S2InvalidTxSZ()
// =======================
// Detect erroneous configuration of stage 2 TxSZ field if the implementation
// does not constrain the value of TxSZ

boolean AArch64.S2InvalidTxSZ(S2TTWParams walkparams, boolean s1aarch64)
    mintxsz = AArch64.S2MinTxSZ(walkparams.ds, walkparams.tgx, s1aarch64);
    maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);

    return UInt(walkparams.txsz) < mintxsz || UInt(walkparams.txsz) > maxtxsz;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.VAIsOutOfRange

```
// AArch64.VAIsOutOfRange()
// ========================
// Check bits not resolved by translation are identical and of accepted value

boolean AArch64.VAIsOutOfRange(bits(64) va, AccType acctype, Regime regime, S1TTWParams walkparams)
    addrtop = AArch64.AddrTop(walkparams.tbid, acctype, walkparams.tbi);
    // Input Address size
    iasize  = AArch64.IASize(walkparams.txsz);

    if HasUnprivileged(regime) then
        if AArch64.GetVARange(va) == VARange_LOWER then
            return !IsZero(va<addrtop:iasize>);
        else
            return !IsOnes(va<addrtop:iasize>);
    else
        return !IsZero(va<addrtop:iasize>);
```

Library pseudocode for aarch64/translation/vmsa_mematTr/AArch64.S2ApplyFWBMemAttrs

```
// AArch64.S2ApplyFWBMemAttrs()
// ============================
// Apply stage 2 forced Write-Back on stage 1 memory attributes.

MemoryAttributes AArch64.S2ApplyFWBMemAttrs(MemoryAttributes s1_mematTrs,
                                            bits(4) s2_attr, bits(2) s2_sh)
    MemoryAttributes mematTrs;

    if s2_attr<2> == '0' then          // S2 Device, S1 any
        s2_device = DecodeDevice(s2_attr<1:0>);
        mematTrs.memtype = MemType_Device;
        if s1_mematTrs.memtype == MemType_Device then
            mematTrs.device = S2CombineS1Device(s1_mematTrs.device, s2_device);
        else
            mematTrs.device = s2_device;

    elsif s2_attr<1:0> == '11' then    // S2 attr = S1 attr
        mematTrs = s1_mematTrs;

    elsif s2_attr<1:0> == '10' then    // Force writeback
        mematTrs.memtype = MemType_Normal;
        mematTrs.inner.attrs = MemAttr_WB;
        mematTrs.outer.attrs = MemAttr_WB;

        if (s1_mematTrs.memtype == MemType_Normal &&
                s1_mematTrs.inner.attrs != MemAttr_NC) then
            mematTrs.inner.hints     = s1_mematTrs.inner.hints;
            mematTrs.inner.transient = s1_mematTrs.inner.transient;
        else
            mematTrs.inner.hints     = MemHint_RWA;
            mematTrs.inner.transient = FALSE;

        if (s1_mematTrs.memtype == MemType_Normal &&
                s1_mematTrs.outer.attrs != MemAttr_NC) then
            mematTrs.outer.hints     = s1_mematTrs.outer.hints;
            mematTrs.outer.transient = s1_mematTrs.outer.transient;
        else
            mematTrs.outer.hints     = MemHint_RWA;
            mematTrs.outer.transient = FALSE;

    else                               // Non-cacheable unless S1 is device
        if s1_mematTrs.memtype == MemType_Device then
            mematTrs = s1_mematTrs;
        else
            MemAttrHints cacheability_attr;
            cacheability_attr.attrs = MemAttr_NC;

            mematTrs.memtype = MemType_Normal;
            mematTrs.inner   = cacheability_attr;
            mematTrs.outer   = cacheability_attr;

    s2_shareability = DecodeShareability(s2_sh);
    mematTrs.shareability = S2CombineS1Shareability(s1_mematTrs.shareability, s2_shareability);
    mematTrs.tagged       = IsS2ResultTagged(mematTrs, s1_mematTrs.tagged);

    mematTrs.shareability = EffectiveShareability(mematTrs);
    return mematTrs;
```

Library pseudocode for aarch64/translation/vmsa_tlbcontext/AArch64.GetS1TLBContext

```
// AArch64.GetS1TLBContext()
// =========================
// Gather translation context for accesses with VA to match against TLB entries

TLBContext AArch64.GetS1TLBContext(Regime regime, SecurityState ss, bits(64) va, TGx tg)
    TLBContext tlbcontext;

    case regime of
        when Regime_EL3  tlbcontext = AArch64.TLBContextEL3(ss, va, tg);
        when Regime_EL2  tlbcontext = AArch64.TLBContextEL2(ss, va, tg);
        when Regime_EL20 tlbcontext = AArch64.TLBContextEL20(ss, va, tg);
        when Regime_EL10 tlbcontext = AArch64.TLBContextEL10(ss, va, tg);

    tlbcontext.includes_s1  = TRUE;
    // The following may be amended for EL1&0 Regime if caching of stage 2 is successful
    tlbcontext.includes_s2  = FALSE;
    // The following may be amended if Granule Protection Check passes
    tlbcontext.includes_gpt = FALSE;
    return tlbcontext;
```

Library pseudocode for aarch64/translation/vmsa_tlbcontext/AArch64.GetS2TLBContext

```
// AArch64.GetS2TLBContext()
// =========================
// Gather translation context for accesses with IPA to match against TLB entries

TLBContext AArch64.GetS2TLBContext(SecurityState ss, FullAddress ipa, TGx tg)
    assert EL2Enabled();

    TLBContext tlbcontext;

    tlbcontext.ss       = ss;
    tlbcontext.regime   = Regime_EL10;
    tlbcontext.ipaspace = ipa.paspace;
    tlbcontext.vmid     = VMID[];
    tlbcontext.tg       = tg;
    tlbcontext.ia       = ZeroExtend(ipa.address, 64);
    if HaveCommonNotPrivateTransExt() then
        tlbcontext.cnp = if ipa.paspace == PAS_Secure then VSTTBR_EL2.CnP else VTTBR_EL2.CnP;
    else
        tlbcontext.cnp = '0';

    tlbcontext.includes_s1  = FALSE;
    tlbcontext.includes_s2  = TRUE;
    // This amy be amended if Granule Protection Check passes
    tlbcontext.includes_gpt = FALSE;
    return tlbcontext;
```

Library pseudocode for aarch64/translation/vmsa_tlbcontext/AArch64.TLBContextEL10

```
// AArch64.TLBContextEL10()
// ========================
// Gather translation context for accesses under EL10 regime to match against TLB entries

TLBContext AArch64.TLBContextEL10(SecurityState ss, bits(64) va, TGx tg)
    TLBContext tlbcontext;

    tlbcontext.ss     = ss;
    tlbcontext.regime = Regime_EL10;
    tlbcontext.vmid   = VMID[];
    tlbcontext.asid   = if TCR_EL1.A1 == '0' then TTBR0_EL1.ASID else TTBR1_EL1.ASID;
    tlbcontext.tg     = tg;
    tlbcontext.ia     = va;

    if HaveCommonNotPrivateTransExt() then
        if AArch64.GetVARange(va) == VARange_LOWER then
            tlbcontext.cnp = TTBR0_EL1.CnP;
        else
            tlbcontext.cnp = TTBR1_EL1.CnP;
    else
        tlbcontext.cnp = '0';

    return tlbcontext;
```

Library pseudocode for aarch64/translation/vmsa_tlbcontext/AArch64.TLBContextEL2

```
// AArch64.TLBContextEL2()
// =======================
// Gather translation context for accesses under EL2 regime to match against TLB entries

TLBContext AArch64.TLBContextEL2(SecurityState ss, bits(64) va, TGx tg)
    TLBContext tlbcontext;

    tlbcontext.ss     = ss;
    tlbcontext.regime = Regime_EL2;
    tlbcontext.tg     = tg;
    tlbcontext.ia     = va;
    tlbcontext.cnp    = if HaveCommonNotPrivateTransExt() then TTBR0_EL2.CnP else '0';

    return tlbcontext;
```

Library pseudocode for aarch64/translation/vmsa_tlbcontext/AArch64.TLBContextEL20

```
// AArch64.TLBContextEL20()
// ========================
// Gather translation context for accesses under EL20 regime to match against TLB entries

TLBContext AArch64.TLBContextEL20(SecurityState ss, bits(64) va, TGx tg)
    TLBContext tlbcontext;

    tlbcontext.ss     = ss;
    tlbcontext.regime = Regime_EL20;
    tlbcontext.asid   = if TCR_EL2.A1 == '0' then TTBR0_EL2.ASID else TTBR1_EL2.ASID;
    tlbcontext.tg     = tg;
    tlbcontext.ia     = va;

    if HaveCommonNotPrivateTransExt() then
        if AArch64.GetVARange(va) == VARange_LOWER then
            tlbcontext.cnp = TTBR0_EL2.CnP;
        else
            tlbcontext.cnp = TTBR1_EL2.CnP;
    else
        tlbcontext.cnp = '0';

    return tlbcontext;
```

Library pseudocode for aarch64/translation/vmsa_tlbcontext/AArch64.TLBContextEL3

```
// AArch64.TLBContextEL3()
// =======================
// Gather translation context for accesses under EL3 regime to match against TLB entries

TLBContext AArch64.TLBContextEL3(SecurityState ss, bits(64) va, TGx tg)
    TLBContext tlbcontext;

    tlbcontext.ss     = ss;
    tlbcontext.regime = Regime_EL3;
    tlbcontext.tg     = tg;
    tlbcontext.ia     = va;
    tlbcontext.cnp    = if HaveCommonNotPrivateTransExt() then TTBR0_EL3.CnP else '0';

    return tlbcontext;
```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64.AccessUsesEL

```
// AArch64.AccessUsesEL()
// ======================
// Returns the Exception Level of the regime that will manage the translation for a given access type.

bits(2) AArch64.AccessUsesEL(AccType acctype)
    if acctype IN {AccType_UNPRIV, AccType_UNPRIVSTREAM} then
        return EL0;
    elsif acctype == AccType_NV2REGISTER then
        return EL2;
    else
        return PSTATE.EL;
```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64.FullTranslate

```
// AArch64.FullTranslate()
// =======================
// Address translation as specified by VMSA
// Alignment check NOT due to memory type is expected to be done before translation

AddressDescriptor AArch64.FullTranslate(bits(64) va, AccType acctype,
                                        boolean iswrite, boolean aligned)

    fault = NoFault();
    fault.acctype = acctype;
    fault.write   = iswrite;

    ispriv = PSTATE.EL != EL0 && !(acctype IN {AccType_UNPRIV, AccType_UNPRIVSTREAM});
    regime = TranslationRegime(PSTATE.EL, acctype);

    ss = SecurityStateAtEL(PSTATE.EL);

    AddressDescriptor ipa;
    (fault, ipa) = AArch64.S1Translate(fault, regime, ss, va, acctype, aligned, iswrite,
                                       ispriv);

    if fault.statuscode != Fault_None then
        return CreateFaultyAddressDescriptor(va, fault);

    assert (ss == SS_Realm) IMPLIES EL2Enabled();
    if regime == Regime_EL10 && EL2Enabled() then
        s1aarch64 = TRUE;
        s2fs1walk = FALSE;
        AddressDescriptor pa;
        (fault, pa) = AArch64.S2Translate(fault, ipa, s1aarch64, ss, s2fs1walk,
                                          acctype, aligned, iswrite, ispriv);

        if fault.statuscode != Fault_None then
            return CreateFaultyAddressDescriptor(va, fault);
        else
            return pa;
    else
        return ipa;
```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64.MemSwapTableDesc

```
// AArch64.MemSwapTableDesc()
// ==========================
// Perform HW update of table descriptor as an atomic operation

(FaultRecord, bits(64)) AArch64.MemSwapTableDesc(FaultRecord fault_in, bits(64) prev_desc,
                                                 bits(64) new_desc, bit ee,
                                                 AddressDescriptor descupdateaddress)
    descupdateaccess = CreateAccessDescriptor(AccType_ATOMICRW);
    FaultRecord fault = fault_in;
    boolean iswrite;

    if HaveRME() then
        fault.gpcf = GranuleProtectionCheck(descupdateaddress, descupdateaccess);
        if fault.gpcf.gpf != GPCF_None then
            fault.statuscode = Fault_GPCFOnWalk;
            fault.paddress   = descupdateaddress.paddress;
            fault.gpcfs2walk = fault.secondstage;
            return (fault, bits(64) UNKNOWN);

    // All observers in the shareability domain observe the
    // following memory read and write accesses atomically.
    (memstatus, mem_desc) = PhysMemRead(descupdateaddress, 8, descupdateaccess);
    if ee == '1' then
        mem_desc = BigEndianReverse(mem_desc);

    if IsFault(memstatus) then
        iswrite = FALSE;
        fault = HandleExternalTTWAbort(memstatus, iswrite, descupdateaddress, descupdateaccess,
                                       8, fault);
        if IsFault(fault.statuscode) then
            fault.acctype = AccType_ATOMICRW;
            return (fault, bits(64) UNKNOWN);

    if mem_desc == prev_desc  then
        ordered_new_desc = if ee == '1' then BigEndianReverse(new_desc) else new_desc;
        memstatus = PhysMemWrite(descupdateaddress, 8, descupdateaccess, ordered_new_desc);

        if IsFault(memstatus) then
            iswrite = TRUE;
            fault = HandleExternalTTWAbort(memstatus, iswrite, descupdateaddress, descupdateaccess,
                                           8, fault);

            if IsFault(fault.statuscode) then
                fault.acctype = AccType_ATOMICRW;
                return (fault, bits(64) UNKNOWN);

        // Reflect what is now in memory (in little endian format)
        mem_desc = new_desc;

    return (fault, mem_desc);
```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64.S1DisabledOutput

```
// AArch64.S1DisabledOutput()
// ==========================
// Map the VA to IPA/PA and assign default memory attributes

(FaultRecord, AddressDescriptor) AArch64.S1DisabledOutput(FaultRecord fault_in, Regime regime,
                                                          SecurityState ss, bits(64) va_in,
                                                          AccType acctype, boolean aligned)

    bits(64) va = va_in;
    walkparams = AArch64.GetS1TTWParams(regime, va);
    FaultRecord fault = fault_in;

    // No memory page is guarded when stage 1 address translation is disabled
    SetInGuardedPage(FALSE);

    // Output Address
    FullAddress oa;
    oa.address = va<51:0>;
    case ss of
        when SS_Secure    oa.paspace = PAS_Secure;
        when SS_NonSecure oa.paspace = PAS_NonSecure;
        when SS_Root      oa.paspace = PAS_Root;
        when SS_Realm     oa.paspace = PAS_Realm;

    MemoryAttributes memattrs;
    if regime == Regime_EL10 && EL2Enabled() && walkparams.dc == '1' then
        MemAttrHints default_cacheability;
        default_cacheability.attrs     = MemAttr_WB;
        default_cacheability.hints     = MemHint_RWA;
        default_cacheability.transient = FALSE;

        memattrs.memtype      = MemType_Normal;
        memattrs.outer        = default_cacheability;
        memattrs.inner        = default_cacheability;
        memattrs.shareability = Shareability_NSH;
        memattrs.tagged       = walkparams.dct == '1';
        memattrs.xs           = '0';
    elsif acctype == AccType_IFETCH then
        MemAttrHints i_cache_attr;
        if AArch64.S1ICacheEnabled(regime) then
            i_cache_attr.attrs     = MemAttr_WT;
            i_cache_attr.hints     = MemHint_RA;
            i_cache_attr.transient = FALSE;
        else
            i_cache_attr.attrs     = MemAttr_NC;

        memattrs.memtype      = MemType_Normal;
        memattrs.outer        = i_cache_attr;
        memattrs.inner        = i_cache_attr;
        memattrs.shareability = Shareability_OSH;
        memattrs.tagged       = FALSE;
        memattrs.xs           = '1';
    else
        memattrs.memtype      = MemType_Device;
        memattrs.device       = DeviceType_nGnRnE;
        memattrs.shareability = Shareability_OSH;
        memattrs.xs           = '1';

    fault.level = 0;
    addrtop     = AArch64.AddrTop(walkparams.tbid, acctype, walkparams.tbi);
    if !IsZero(va<addrtop:AArch64.PAMax()>) then
        fault.statuscode = Fault_AddressSize;
    elsif AArch64.S1HasAlignmentFault(acctype, aligned, walkparams.ntlsmd, memattrs) then
        fault.statuscode = Fault_Alignment;

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);
    else
        ipa = CreateAddressDescriptor(va_in, oa, memattrs);
        return (fault, ipa);
```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64.S1Translate

```
// AArch64.S1Translate()
// =====================
// Translate VA to IPA/PA depending on the regime

(FaultRecord, AddressDescriptor) AArch64.S1Translate(FaultRecord fault_in, Regime regime,
                                                    SecurityState ss, bits(64) va,
                                                    AccType acctype, boolean aligned_in,
                                                    boolean iswrite_in,
                                                    boolean ispriv)
    FaultRecord fault = fault_in;
    boolean aligned = aligned_in;
    boolean iswrite = iswrite_in;
    // Prepare fault fields in case a fault is detected
    fault.secondstage = FALSE;
    fault.s2fs1walk   = FALSE;

    if !AArch64.S1Enabled(regime, acctype) then
        return AArch64.S1DisabledOutput(fault, regime, ss, va, acctype, aligned);

    walkparams = AArch64.GetS1TTWParams(regime, va);

    if (AArch64.S1InvalidTxSZ(walkparams) ||
            (!ispriv && walkparams.e0pd == '1') ||
            (!ispriv && walkparams.nfd == '1' && IsDataAccess(acctype) && TSTATE.depth > 0) ||
            (!ispriv && walkparams.nfd == '1' && acctype == AccType_NONFAULT) ||
            AArch64.VAIsOutOfRange(va, acctype, regime, walkparams)) then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN);

    AddressDescriptor descaddress;
    TTWState walkstate;
    bits(64) descriptor;
    bits(64) new_desc;
    bits(64) mem_desc;
    repeat
        (fault, descaddress, walkstate, descriptor) = AArch64.S1Walk(fault, walkparams, va, regime,
                                                                     ss, acctype, iswrite, ispriv);

        if fault.statuscode != Fault_None then
            return (fault, AddressDescriptor UNKNOWN);

        if acctype == AccType_IFETCH then
            // Flag the fetched instruction is from a guarded page
            SetInGuardedPage(walkstate.guardedpage == '1');

        if AArch64.S1HasAlignmentFault(acctype, aligned, walkparams.ntlsmd,
                                       walkstate.memattrs) then
            fault.statuscode = Fault_Alignment;
        elsif IsAtomicRW(acctype) then
            if AArch64.S1HasPermissionsFault(regime, ss, walkstate, walkparams,
                                             ispriv, acctype, FALSE) then
                // The Permission fault was not caused by lack of write permissions
                fault.statuscode = Fault_Permission;
                fault.write      = FALSE;
            elsif AArch64.S1HasPermissionsFault(regime, ss, walkstate, walkparams,
                                                ispriv, acctype, TRUE) then
                // The Permission fault was caused by lack of write permissions
                fault.statuscode = Fault_Permission;
                fault.write      = TRUE;
        elsif AArch64.S1HasPermissionsFault(regime, ss, walkstate, walkparams,
                                            ispriv, acctype, iswrite) then
            fault.statuscode = Fault_Permission;

        new_desc = descriptor;
        if walkparams.ha == '1' && AArch64.SettingAccessFlagPermitted(fault) then
            // Set descriptor AF bit
            new_desc<10> = '1';

        // If HW update of dirty bit is enabled, the walk state permissions
```

```
        // will already reflect a configuration permitting writes.
        // The update of the descriptor occurs only if the descriptor bits in
        // memory do not reflect that and the access instigates a write.
        if (AArch64.SettingDirtyStatePermitted(fault) &&
                walkparams.ha  == '1' &&
                walkparams.hd  == '1' &&
                descriptor<51> == '1' && // Descriptor DBM bit
                (IsAtomicRW(acctype) || iswrite) &&
                !(acctype IN {AccType_AT, AccType_ATPAN, AccType_IC, AccType_DC})) then
            // Clear descriptor AP[2] bit permitting stage 1 writes
            new_desc<7> = '0';

        AddressDescriptor descupdateaddress;
        FaultRecord s2fault;
        // Either the access flag was clear or AP<2> is set
        if new_desc != descriptor then
            if regime == Regime_EL10 && EL2Enabled() then
                s1aarch64 = TRUE;
                s2fs1walk = TRUE;
                aligned   = TRUE;
                iswrite   = TRUE;
                (s2fault, descupdateaddress) = AArch64.S2Translate(fault, descaddress, s1aarch64,
                                                                   ss, s2fs1walk, AccType_ATOMICRW,
                                                                   aligned, iswrite,
                                                                   ispriv);

                if s2fault.statuscode != Fault_None then
                    return (s2fault, AddressDescriptor UNKNOWN);
            else
                descupdateaddress = descaddress;

            (fault, mem_desc) = AArch64.MemSwapTableDesc(fault, descriptor, new_desc,
                                                         walkparams.ee, descupdateaddress);

    until new_desc == descriptor || mem_desc == new_desc;

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);

    // Output Address
    oa = StageOA(va, walkparams.tgx, walkstate);
    MemoryAttributes mematTrs;
    if (acctype == AccType_IFETCH &&
        (walkstate.mematTrs.memtype == MemType_Device || !AArch64.S1ICacheEnabled(regime))) then
        // Treat memory attributes as Normal Non-Cacheable
        mematTrs = NormalNCMemAttr();
        mematTrs.xs = walkstate.mematTrs.xs;
    elsif (acctype != AccType_IFETCH && !AArch64.S1DCacheEnabled(regime) &&
             walkstate.mematTrs.memtype == MemType_Normal) then
        // Treat memory attributes as Normal Non-Cacheable
        mematTrs = NormalNCMemAttr();
        mematTrs.xs = walkstate.mematTrs.xs;

        // The effect of SCTLR_ELx.C when '0' is Constrained UNPREDICTABLE
        // on the Tagged attribute
        if HaveMTE2Ext() && walkstate.mematTrs.tagged then
            mematTrs.tagged = ConstrainUnpredictableBool(Unpredictable_S1CTAGGED);
    else
        mematTrs = walkstate.mematTrs;

    // Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
    // to be either effective value or descriptor value
    if (regime == Regime_EL10 && EL2Enabled() && HCR_EL2.VM == '1' &&
            !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
        mematTrs.shareability = walkstate.mematTrs.shareability;
    else
        mematTrs.shareability = EffectiveShareability(mematTrs);

    if acctype == AccType_ATOMICLS64 && mematTrs.memtype == MemType_Normal then
        if mematTrs.inner.attrs != MemAttr_NC || mematTrs.outer.attrs != MemAttr_NC then
```

```
            fault.statuscode = Fault_Exclusive;
            return (fault, AddressDescriptor UNKNOWN);

    ipa = CreateAddressDescriptor(va, oa, memattrs);
    return (fault, ipa);
```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64.S2Translate

```
// AArch64.S2Translate()
// =====================
// Translate stage 1 IPA to PA and combine memory attributes

(FaultRecord, AddressDescriptor) AArch64.S2Translate(FaultRecord fault_in, AddressDescriptor ipa,
                                                    boolean s1aarch64, SecurityState ss,
                                                    boolean s2fs1walk, AccType acctype,
                                                    boolean aligned, boolean iswrite,
                                                     boolean ispriv)
    walkparams = AArch64.GetS2TTWParams(ss, ipa.paddress.paspace, s1aarch64);
    FaultRecord fault = fault_in;

    // Prepare fault fields in case a fault is detected
    fault.statuscode  = Fault_None; // Ignore any faults from stage 1
    fault.secondstage = TRUE;
    fault.s2fs1walk   = s2fs1walk;
    fault.ipaddress   = ipa.paddress;

    if walkparams.vm != '1' then
        // Stage 2 translation is disabled
        return (fault, ipa);

    if (AArch64.S2InvalidTxSZ(walkparams, s1aarch64) ||
            AArch64.S2InvalidSL(walkparams) ||
            AArch64.S2InconsistentSL(walkparams) ||
            AArch64.IPAIsOutOfRange(ipa.paddress.address, walkparams)) then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN);

    AddressDescriptor descaddress;
    TTWState walkstate;
    bits(64) descriptor;
    bits(64) new_desc;
    bits(64) mem_desc;
    repeat
        (fault, descaddress, walkstate, descriptor) = AArch64.S2Walk(fault, ipa, walkparams, ss,
                                                                     acctype, iswrite, s1aarch64);

        if fault.statuscode != Fault_None then
            return (fault, AddressDescriptor UNKNOWN);

        if AArch64.S2HasAlignmentFault(acctype, aligned, walkstate.memattrs) then
            fault.statuscode = Fault_Alignment;
        elsif IsAtomicRW(acctype) then
            if AArch64.S2HasPermissionsFault(s2fs1walk, walkstate, ss, walkparams,
                                             ispriv, acctype, FALSE) then
                // The Permission fault was not caused by lack of write permissions
                fault.statuscode = Fault_Permission;
                fault.write      = FALSE;
            elsif AArch64.S2HasPermissionsFault(s2fs1walk, walkstate, ss, walkparams,
                                                ispriv, acctype, TRUE) then
                // The Permission fault was caused by lack of write permissions.
                // However, HW updates, which are atomic writes for stage 1
                // descriptors, permissions fault reflect the original access.
                fault.statuscode = Fault_Permission;
                if !fault.s2fs1walk then
                    fault.write  = TRUE;
        elsif AArch64.S2HasPermissionsFault(s2fs1walk, walkstate, ss, walkparams,
                                            ispriv, acctype, iswrite) then
            fault.statuscode = Fault_Permission;
        new_desc = descriptor;
        if walkparams.ha == '1' && AArch64.SettingAccessFlagPermitted(fault) then
            // Set descriptor AF bit
            new_desc<10> = '1';

        // If HW update of dirty bit is enabled, the walk state permissions
        // will already reflect a configuration permitting writes.
        // The update of the descriptor occurs only if the descriptor bits in
        // memory do not reflect that and the access instigates a write.
```

```
        if (AArch64.SettingDirtyStatePermitted(fault) &&
                walkparams.ha  == '1' &&
                walkparams.hd  == '1' &&
                descriptor<51> == '1' && // Descriptor DBM bit
                (IsAtomicRW(acctype) || iswrite) &&
                !(acctype IN {AccType_AT, AccType_ATPAN, AccType_IC, AccType_DC})) then
            // Set descriptor S2AP[1] bit permitting stage 2 writes
            new_desc<7> = '1';

        // Either the access flag was clear or S2AP<1> is clear
        if new_desc != descriptor then
            (fault, mem_desc) = AArch64.MemSwapTableDesc(fault, descriptor, new_desc,
                                                         walkparams.ee, descaddress);

    until new_desc == descriptor || mem_desc == new_desc;

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);

    ipa_64 = ZeroExtend(ipa.paddress.address, 64);
    // Output Address
    oa = StageOA(ipa_64, walkparams.tgx, walkstate);
    MemoryAttributes s2_mematTrs;
    if ((s2fs1walk &&
            walkstate.mematTrs.memtype == MemType_Device && walkparams.ptw == '0') ||
        (acctype == AccType_IFETCH &&
            (walkstate.mematTrs.memtype == MemType_Device || HCR_EL2.ID == '1')) ||
        (acctype != AccType_IFETCH &&
             walkstate.mematTrs.memtype == MemType_Normal && HCR_EL2.CD == '1')) then
        // Treat memory attributes as Normal Non-Cacheable
        s2_mematTrs = NormalNCMemAttr();
        s2_mematTrs.xs = walkstate.mematTrs.xs;
    else
        s2_mematTrs = walkstate.mematTrs;

    if !s2fs1walk && acctype == AccType_ATOMICLS64 && s2_mematTrs.memtype == MemType_Normal then
        if s2_mematTrs.inner.attrs != MemAttr_NC || s2_mematTrs.outer.attrs != MemAttr_NC then
            fault.statuscode = Fault_Exclusive;
            return (fault, AddressDescriptor UNKNOWN);

    MemoryAttributes mematTrs;
    if walkparams.fwb == '0' then
        mematTrs = S2CombineS1MemAttrs(ipa.mematTrs, s2_mematTrs);
    else
        mematTrs = s2_mematTrs;

    pa = CreateAddressDescriptor(ipa.vaddress, oa, mematTrs);
    return (fault, pa);
```

Library pseudocode for aarch64/translation/vmsa_translation/ AArch64.SettingAccessFlagPermitted

```
// AArch64.SettingAccessFlagPermitted()
// ====================================
// Determine whether the access flag could be set by HW given the fault status

boolean AArch64.SettingAccessFlagPermitted(FaultRecord fault)
    if fault.statuscode == Fault_None then
        return TRUE;
    elsif fault.statuscode IN {Fault_Alignment, Fault_Permission} then
        return ConstrainUnpredictableBool(Unpredictable_AFUPDATE);
    else
        return FALSE;
```

Library pseudocode for aarch64/translation/vmsa_translation/ AArch64.SettingDirtyStatePermitted

```
// AArch64.SettingDirtyStatePermitted()
// ====================================
// Determine whether the dirty state could be set by HW given the fault status

boolean AArch64.SettingDirtyStatePermitted(FaultRecord fault)
    if fault.statuscode == Fault_None then
        return TRUE;
    elsif fault.statuscode == Fault_Alignment then
        return ConstrainUnpredictableBool(Unpredictable_DBUPDATE);
    else
        return FALSE;
```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64.TranslateAddress

```
// AArch64.TranslateAddress()
// ==========================
// Main entry point for translating an address

AddressDescriptor AArch64.TranslateAddress(bits(64) va, AccType acctype, boolean iswrite,
                                           boolean aligned, integer size)
    result = AArch64.FullTranslate(va, acctype, iswrite, aligned);

    if !IsFault(result) && acctype != AccType_IFETCH then
        result.fault = AArch64.CheckDebug(va, acctype, iswrite, size);

    if HaveRME() && !IsFault(result) && (acctype != AccType_DC ||
        boolean IMPLEMENTATION_DEFINED "GPC Fault on DC operations") then
        accdesc = CreateAccessDescriptor(acctype);
        result.fault.gpcf = GranuleProtectionCheck(result, accdesc);

        if result.fault.gpcf.gpf != GPCF_None then
            result.fault.statuscode = Fault_GPCFOnOutput;
            result.fault.paddress   = result.paddress;
            result.fault.acctype    = acctype;
            result.fault.write      = iswrite;

    if !IsFault(result) && acctype == AccType_IFETCH then
        result.fault = AArch64.CheckDebug(va, acctype, iswrite, size);

    // Update virtual address for abort functions
    result.vaddress = ZeroExtend(va, 64);

    return result;
```

Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64.BlockDescSupported

```
// AArch64.BlockDescSupported()
// ============================
// Determine whether a block descriptor is valid for the given granule size
// and level

boolean AArch64.BlockDescSupported(bit ds, TGx tgx, integer level)
    case tgx of
        when TGx_4KB  return level == 2 || level == 1 || (level == 0 && ds == '1');
        when TGx_16KB return level == 2 || (level == 1 && ds == '1');
        when TGx_64KB return level == 2 || (level == 1 && AArch64.PAMax() == 52);

    return FALSE;
```

Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64.BlocknTFaults

```
// AArch64.BlocknTFaults()
// =======================
// Identify whether the nT bit in a block descriptor is effectively set
// causing a translation fault

boolean AArch64.BlocknTFaults(bits(64) descriptor)
    if !HaveBlockBBM() then
        return FALSE;

    bbm_level = AArch64.BlockBBMSupportLevel();
    nT_faults = boolean IMPLEMENTATION_DEFINED "BBM level 1 or 2 support nT bit causes Translation Fault";

    return bbm_level IN {1, 2} && descriptor<16> == '1' && nT_faults;
```

Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64.ContiguousBit

```
// AArch64.ContiguousBit()
// =======================
// Get the value of the contiguous bit

bit AArch64.ContiguousBit(TGx tgx, integer level, bits(64) descriptor)
    // When using TGx 64KB and FEAT_LPA is implememted,
    // the Contiguous bit is RES0 for Block descriptors at level 1

    if tgx == TGx_64KB && level == 1 then
        return '0'; // RES0

    // When the effective value of TCR_ELx.DS is '1',
    // the Contiguous bit is RES0 for all the following:
    //      * For TGx 4KB, Block descriptors at level 0
    //      * For TGx 16KB, Block descriptors at level 1

    if tgx == TGx_16KB && level == 1 then
        return '0'; // RES0

    if tgx == TGx_4KB  && level == 0 then
        return '0'; // RES0

    return descriptor<52>;
```

Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64.DecodeDescriptorType

```
// AArch64.DecodeDescriptorType()
// ==============================
// Determine whether the descriptor is a page, block or table

DescriptorType AArch64.DecodeDescriptorType(bits(64) descriptor, bit ds,
                                            TGx tgx, integer level)
    if descriptor<1:0> == '11' && level == FINAL_LEVEL then
        return DescriptorType_Page;
    elsif descriptor<1:0> == '11' then
        return DescriptorType_Table;
    elsif descriptor<1:0> == '01' then
        if AArch64.BlockDescSupported(ds, tgx, level) then
            return DescriptorType_Block;
        else
            return DescriptorType_Invalid;
    else
        return DescriptorType_Invalid;
```

Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64.S1ApplyOutputPerms

```
// AArch64.S1ApplyOutputPerms()
// ============================
// Apply output permissions encoded in stage 1 page/block descriptors

Permissions AArch64.S1ApplyOutputPerms(Permissions permissions_in, bits(64) descriptor,
                                       Regime regime, S1TTWParams walkparams)
    Permissions permissions = permissions_in;
    if regime == Regime_EL10 && EL2Enabled() && walkparams.nv1 == '1' then
        permissions.ap<2:1> = descriptor<7>:'0';
        permissions.pxn     = descriptor<54>;
    elsif HasUnprivileged(regime) then
        permissions.ap<2:1> = descriptor<7:6>;
        permissions.uxn     = descriptor<54>;
        permissions.pxn     = descriptor<53>;
    else
        permissions.ap<2:1> = descriptor<7>:'1';
        permissions.xn      = descriptor<54>;

    // Descriptors marked with DBM set have the effective value of AP[2] cleared.
    // This implies no Permission faults caused by lack of write permissions are
    // reported, and the Dirty bit can be set.
    if walkparams.ha == '1' && walkparams.hd == '1' && descriptor<51> == '1' then
        permissions.ap<2> = '0';

    return permissions;
```

Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64.S1ApplyTablePerms

```
// AArch64.S1ApplyTablePerms()
// ===========================
// Apply hierarchical permissions encoded in stage 1 table descriptors

Permissions AArch64.S1ApplyTablePerms(Permissions permissions_in, bits(64) descriptor,
                                      Regime regime, S1TTWParams walkparams)
    Permissions permissions = permissions_in;
    bits(2) ap_table;
    bit pxn_table;
    bit uxn_table;
    bit xn_table;
    if regime == Regime_EL10 && EL2Enabled() && walkparams.nv1 == '1' then
        ap_table  = descriptor<62>:'0';
        pxn_table = descriptor<60>;
        permissions.ap_table  = permissions.ap_table  OR ap_table;
        permissions.pxn_table = permissions.pxn_table OR pxn_table;

    elsif HasUnprivileged(regime) then
        ap_table  = descriptor<62:61>;
        uxn_table = descriptor<60>;
        pxn_table = descriptor<59>;
        permissions.ap_table  = permissions.ap_table  OR ap_table;
        permissions.uxn_table = permissions.uxn_table OR uxn_table;
        permissions.pxn_table = permissions.pxn_table OR pxn_table;
    else
        ap_table = descriptor<62>:'0';
        xn_table = descriptor<60>;
        permissions.ap_table = permissions.ap_table OR ap_table;
        permissions.xn_table = permissions.xn_table OR xn_table;

    return permissions;
```

Library pseudocode for aarch64/translation/vmsa_ttentry/AArch64.S2ApplyOutputPerms

```
// AArch64.S2ApplyOutputPerms()
// ============================
// Apply output permissions encoded in stage 2 page/block descriptors

Permissions AArch64.S2ApplyOutputPerms(bits(64) descriptor, S2TTWParams walkparams)
    Permissions permissions;

    permissions.s2ap = descriptor<7:6>;
    permissions.s2xn = descriptor<54>;

    if HaveExtendedExecuteNeverExt() then
        permissions.s2xnx = descriptor<53>;
    else
        permissions.s2xnx = '0';

    // Descriptors marked with DBM set have the effective value of S2AP[1] set.
    // This implies no Permission faults caused by lack of write permissions are
    // reported, and the Dirty bit can be set.
    if walkparams.ha == '1' && walkparams.hd == '1' && descriptor<51> == '1' then
        permissions.s2ap<1> = '1';

    return permissions;
```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.S1InitialTTWState

```
// AArch64.S1InitialTTWState()
// ===========================
// Set properties of first access to translation tables in stage 1

TTWState AArch64.S1InitialTTWState(S1TTWParams walkparams, bits(64) va, Regime regime,
                                   SecurityState ss)
    TTWState    walkstate;
    FullAddress tablebase;
    Permissions permissions;

    startlevel  = AArch64.S1StartLevel(walkparams);
    ttbr        = AArch64.S1TTBR(regime, va);
    case ss of
        when SS_Secure    tablebase.paspace = PAS_Secure;
        when SS_NonSecure tablebase.paspace = PAS_NonSecure;
        when SS_Root      tablebase.paspace = PAS_Root;
        when SS_Realm     tablebase.paspace = PAS_Realm;

    tablebase.address = AArch64.TTBaseAddress(ttbr, walkparams.txsz, walkparams.ps, walkparams.ds,
                                              walkparams.tgx, startlevel);

    permissions.ap_table = '00';
    if HasUnprivileged(regime) then
        permissions.uxn_table = '0';
        permissions.pxn_table = '0';
    else
        permissions.xn_table  = '0';

    walkstate.baseaddress = tablebase;
    walkstate.level       = startlevel;
    walkstate.istable     = TRUE;
    // In regimes that support global and non-global translations, translation
    // table entries from lookup levels other than the final level of lookup
    // are treated as being non-global
    walkstate.nG          = if HasUnprivileged(regime) then '1' else '0';
    walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);
    walkstate.permissions = permissions;

    return walkstate;
```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.S1NextWalkStateLast

```
// AArch64.S1NextWalkStateLast()
// =============================
// Decode stage 1 page or block descriptor as output to this stage of translation

TTWState AArch64.S1NextWalkStateLast(TTWState currentstate, Regime regime, SecurityState ss,
                                     S1TTWParams walkparams, bits(64) descriptor)
    TTWState    nextstate;
    FullAddress baseaddress;

    if currentstate.level == FINAL_LEVEL then
        baseaddress.address = AArch64.PageBase(descriptor, walkparams.ds, walkparams.tgx);
    else
        baseaddress.address = AArch64.BlockBase(descriptor, walkparams.ds, walkparams.tgx,
                                                currentstate.level);

    if currentstate.baseaddress.paspace == PAS_Secure then
        // Determine PA space of the block from NS bit
        baseaddress.paspace = if descriptor<5> == '0' then PAS_Secure else PAS_NonSecure;
    elsif currentstate.baseaddress.paspace == PAS_Root then
        // Determine PA space of the block from NSE and NS bits
        case descriptor<11,5> of
            when '00' baseaddress.paspace = PAS_Secure;
            when '01' baseaddress.paspace = PAS_NonSecure;
            when '10' baseaddress.paspace = PAS_Root;
            when '11' baseaddress.paspace = PAS_Realm;
    elsif (currentstate.baseaddress.paspace == PAS_Realm &&
            regime IN {Regime_EL2, Regime_EL20}) then
        // Realm EL2 and EL2&0 regimes have a stage 1 NS bit
        baseaddress.paspace = if descriptor<5> == '0' then PAS_Realm else PAS_NonSecure;
    elsif currentstate.baseaddress.paspace == PAS_Realm then
        // Realm EL1&0 regime does not have a stage 1 NS bit
        baseaddress.paspace = PAS_Realm;
    else
        baseaddress.paspace = PAS_NonSecure;

    nextstate.istable     = FALSE;
    nextstate.level       = currentstate.level;
    nextstate.baseaddress = baseaddress;

    attrindx = descriptor<4:2>;
    sh = if walkparams.ds == '1' then walkparams.sh else descriptor<9:8>;
    attr = MAIRAttr(UInt(attrindx), walkparams.mair);
    s1aarch64 = TRUE;

    nextstate.memattrs    = S1DecodeMemAttrs(attr, sh, s1aarch64);
    nextstate.permissions = AArch64.S1ApplyOutputPerms(currentstate.permissions, descriptor,
                                                       regime, walkparams);
    nextstate.contiguous  = AArch64.ContiguousBit(walkparams.tgx, currentstate.level, descriptor);

    if !HasUnprivileged(regime) then
        nextstate.nG = '0';
    elsif ss == SS_Secure && currentstate.baseaddress.paspace == PAS_NonSecure then
        // In Secure state, a translation must be treated as non-global,
        // regardless of the value of the nG bit,
        // if NSTable is set to 1 at any level of the translation table walk
        nextstate.nG = '1';
    else
        nextstate.nG = descriptor<11>;

    nextstate.guardedpage = descriptor<50>;

    return nextstate;
```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.S1NextWalkStateTable

```
// AArch64.S1NextWalkStateTable()
// ==============================
// Decode stage 1 table descriptor to transition to the next level

TTWState AArch64.S1NextWalkStateTable(TTWState currentstate, Regime regime, S1TTWParams walkparams,
                                      bits(64) descriptor)
    TTWState    nextstate;
    FullAddress tablebase;

    tablebase.address = AArch64.NextTableBase(descriptor, walkparams.ds, walkparams.tgx);
    if currentstate.baseaddress.paspace == PAS_Secure then
        // Determine PA space of the next table from NSTable bit
        tablebase.paspace = if descriptor<63> == '0' then PAS_Secure else PAS_NonSecure;
    else
        // Otherwise bit 63 is RES0 and there is no NSTable bit
        tablebase.paspace = currentstate.baseaddress.paspace;

    nextstate.istable     = TRUE;
    nextstate.nG          = currentstate.nG;
    nextstate.level       = currentstate.level + 1;
    nextstate.baseaddress = tablebase;
    nextstate.memattrs    = currentstate.memattrs;

    if walkparams.hpd == '0' then
        nextstate.permissions = AArch64.S1ApplyTablePerms(currentstate.permissions, descriptor,
                                                          regime, walkparams);
    else
        nextstate.permissions = currentstate.permissions;

    return nextstate;
```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.S1Walk

```
// AArch64.S1Walk()
// ================
// Traverse stage 1 translation tables obtaining the final descriptor
// as well as the address leading to that descriptor

(FaultRecord, AddressDescriptor, TTWState, bits(64)) AArch64.S1Walk(FaultRecord fault_in,
        S1TTWParams walkparams, bits(64) va, Regime regime, SecurityState ss,
        AccType acctype, boolean iswrite_in, boolean ispriv)
    FaultRecord fault = fault_in;
    boolean iswrite = iswrite_in;
    if HasUnprivileged(regime) && AArch64.S1EPD(regime, va) == '1' then
        fault.statuscode = Fault_Translation;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

    walkstate = AArch64.S1InitialTTWState(walkparams, va, regime, ss);

    // Detect Address Size Fault by TTB
    if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, va) then
        fault.statuscode = Fault_AddressSize;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

    bits(64) descriptor;
    AddressDescriptor walkaddress;

    walkaddress.vaddress = va;
    if !AArch64.S1DCacheEnabled(regime) then
        walkaddress.memattrs = NormalNCMemAttr();
        walkaddress.memattrs.xs = walkstate.memattrs.xs;
    else
        walkaddress.memattrs = walkstate.memattrs;

    // Shareability value of stage 1 translation subject to stage 2 is IMPLEMENTATION DEFINED
    // to be either effective value or descriptor value
    if (regime == Regime_EL10 && EL2Enabled() && HCR_EL2.VM == '1' &&
            !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
        walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
    else
        walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

    DescriptorType desctype;
    repeat
        fault.level = walkstate.level;
        FullAddress descaddress = AArch64.TTEntryAddress(walkstate.level, walkparams.tgx,
                                                         walkparams.txsz, va,
                                                         walkstate.baseaddress);

        walkaddress.paddress = descaddress;

        if regime == Regime_EL10 && EL2Enabled() then
            s1aarch64 = TRUE;
            s2fs1walk = TRUE;
            aligned   = TRUE;
            iswrite   = FALSE;
            (s2fault, s2walkaddress) = AArch64.S2Translate(fault, walkaddress, s1aarch64, ss,
                                                           s2fs1walk, AccType_TTW, aligned,
                                                           iswrite, ispriv);

            if s2fault.statuscode != Fault_None then
                return (s2fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

            (fault, descriptor) = FetchDescriptor(walkparams.ee, s2walkaddress, fault, 64);
        else
            (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, fault, 64);

        if fault.statuscode != Fault_None then
            return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

        desctype = AArch64.DecodeDescriptorType(descriptor, walkparams.ds, walkparams.tgx,
```

```
                                            walkstate.level);

        case desctype of
            when DescriptorType_Table
                walkstate = AArch64.S1NextWalkStateTable(walkstate, regime, walkparams,
                                                         descriptor);

                // Detect Address Size Fault by table descriptor
                if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, va) then
                    fault.statuscode = Fault_AddressSize;
                    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

            when DescriptorType_Page, DescriptorType_Block
                walkstate = AArch64.S1NextWalkStateLast(walkstate, regime, ss,
                                                        walkparams, descriptor);

            when DescriptorType_Invalid
                fault.statuscode = Fault_Translation;
                return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

            otherwise
                Unreachable();

    until desctype IN {DescriptorType_Page, DescriptorType_Block};

    if (walkstate.contiguous == '1' &&
            AArch64.ContiguousBitFaults(walkparams.txsz, walkparams.tgx, walkstate.level)) then
        fault.statuscode = Fault_Translation;
    elsif desctype == DescriptorType_Block && AArch64.BlocknTFaults(descriptor) then
        fault.statuscode = Fault_Translation;
    // Detect Address Size Fault by final output
    elsif AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, va) then
        fault.statuscode = Fault_AddressSize;
    // Check descriptor AF bit
    elsif (descriptor<10> == '0' && walkparams.ha == '0' &&
            !(acctype IN {AccType_DC, AccType_IC} &&
            !boolean IMPLEMENTATION_DEFINED "Generate access flag fault on IC/DC operations")) then
        fault.statuscode = Fault_AccessFlag;

    return (fault, walkaddress, walkstate, descriptor);
```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.S2InitialTTWState

```
// AArch64.S2InitialTTWState()
// ===========================
// Set properties of first access to translation tables in stage 2

TTWState AArch64.S2InitialTTWState(SecurityState ss, S2TTWParams walkparams)
    TTWState    walkstate;
    FullAddress tablebase;

    ttbr              = VTTBR_EL2;
    startlevel        = AArch64.S2StartLevel(walkparams);
    case ss of
        when SS_NonSecure tablebase.paspace = PAS_NonSecure;
        when SS_Realm     tablebase.paspace = PAS_Realm;
    tablebase.address = AArch64.TTBaseAddress(ttbr, walkparams.txsz, walkparams.ps, walkparams.ds,
                                              walkparams.tgx, startlevel);

    walkstate.baseaddress = tablebase;
    walkstate.level       = startlevel;
    walkstate.istable     = TRUE;
    walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);

    return walkstate;
```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.S2NextWalkStateLast

```
// AArch64.S2NextWalkStateLast()
// =============================
// Decode stage 2 page or block descriptor as output to this stage of translation

TTWState AArch64.S2NextWalkStateLast(TTWState currentstate, SecurityState ss,
                                     S2TTWParams walkparams, AddressDescriptor ipa,
                                     bits(64) descriptor)
    TTWState    nextstate;
    FullAddress baseaddress;

    if ss == SS_Secure then
        baseaddress.paspace = AArch64.SS2OutputPASpace(walkparams, ipa.paddress.paspace);
    elsif ss == SS_Realm then
        if descriptor<55> == '1' then
            baseaddress.paspace = PAS_NonSecure;
        else
            baseaddress.paspace = PAS_Realm;
    else
        baseaddress.paspace = PAS_NonSecure;

    if currentstate.level == FINAL_LEVEL then
        baseaddress.address = AArch64.PageBase(descriptor, walkparams.ds, walkparams.tgx);
    else
        baseaddress.address = AArch64.BlockBase(descriptor, walkparams.ds, walkparams.tgx,
                                                currentstate.level);

    nextstate.istable     = FALSE;
    nextstate.level       = currentstate.level;
    nextstate.baseaddress = baseaddress;
    nextstate.permissions = AArch64.S2ApplyOutputPerms(descriptor, walkparams);

    s2_attr = descriptor<5:2>;
    s2_sh   = if walkparams.ds == '1' then walkparams.sh else descriptor<9:8>;
    s2_fnxs = descriptor<11>;
    if walkparams.fwb == '1' then
        nextstate.memattrs = AArch64.S2ApplyFWBMemAttrs(ipa.memattrs, s2_attr, s2_sh);
        if s2_attr<1:0> == '10' then    // Force writeback
            nextstate.memattrs.xs = '0';
        else
            nextstate.memattrs.xs = if s2_fnxs == '1' then '0' else ipa.memattrs.xs;
    else
        s2aarch64 = TRUE;
        nextstate.memattrs = S2DecodeMemAttrs(s2_attr, s2_sh, s2aarch64);
        nextstate.memattrs.xs = if s2_fnxs == '1' then '0' else ipa.memattrs.xs;
    nextstate.contiguous = AArch64.ContiguousBit(walkparams.tgx, currentstate.level, descriptor);

    return nextstate;
```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.S2NextWalkStateTable

```
// AArch64.S2NextWalkStateTable()
// ==============================
// Decode stage 2 table descriptor to transition to the next level

TTWState AArch64.S2NextWalkStateTable(TTWState currentstate, S2TTWParams walkparams,
                                      bits(64) descriptor)
    TTWState    nextstate;
    FullAddress tablebase;

    tablebase.address = AArch64.NextTableBase(descriptor, walkparams.ds, walkparams.tgx);
    tablebase.paspace = currentstate.baseaddress.paspace;

    nextstate.istable     = TRUE;
    nextstate.level       = currentstate.level + 1;
    nextstate.baseaddress = tablebase;
    nextstate.memattrs    = currentstate.memattrs;

    return nextstate;
```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.S2Walk

```
// AArch64.S2Walk()
// ================
// Traverse stage 2 translation tables obtaining the final descriptor
// as well as the address leading to that descriptor

(FaultRecord, AddressDescriptor, TTWState, bits(64)) AArch64.S2Walk(
        FaultRecord fault_in, AddressDescriptor ipa, S2TTWParams walkparams, SecurityState ss,
        AccType acctype, boolean iswrite, boolean s1aarch64)

    FaultRecord fault = fault_in;
    ipa_64 = ZeroExtend(ipa.paddress.address, 64);

    TTWState walkstate;
    if ss == SS_Secure then
        walkstate = AArch64.SS2InitialTTWState(walkparams, ipa.paddress.paspace);
    else
        walkstate = AArch64.S2InitialTTWState(ss, walkparams);

    // Detect Address Size Fault by TTB
    if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, ipa_64) then
        fault.statuscode = Fault_AddressSize;
        fault.level      = 0;
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

    bits(64) descriptor;
    AddressDescriptor walkaddress;

    walkaddress.vaddress = ipa.vaddress;
    if HCR_EL2.CD == '1' then
        walkaddress.memattrs = NormalNCMemAttr();
        walkaddress.memattrs.xs = walkstate.memattrs.xs;
    else
        walkaddress.memattrs = walkstate.memattrs;

    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

    DescriptorType desctype;
    repeat
        fault.level = walkstate.level;

        FullAddress descaddress;
        if walkstate.level == AArch64.S2StartLevel(walkparams) then
            // Initial lookup might index into concatenated tables
            descaddress = AArch64.S2SLTTEntryAddress(walkparams, ipa.paddress.address,
                                                     walkstate.baseaddress);
        else
            ipa_64 = ZeroExtend(ipa.paddress.address, 64);
            descaddress = AArch64.TTEntryAddress(walkstate.level, walkparams.tgx, walkparams.txsz,
                                                 ipa_64, walkstate.baseaddress);

        walkaddress.paddress = descaddress;
        (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, fault, 64);

        if fault.statuscode != Fault_None then
            return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

        desctype = AArch64.DecodeDescriptorType(descriptor, walkparams.ds, walkparams.tgx,
                                                walkstate.level);

        case desctype of
            when DescriptorType_Table
                walkstate = AArch64.S2NextWalkStateTable(walkstate, walkparams, descriptor);

                // Detect Address Size Fault by table descriptor
                if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, ipa_64) then
                    fault.statuscode = Fault_AddressSize;
                    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

            when DescriptorType_Page, DescriptorType_Block
                walkstate = AArch64.S2NextWalkStateLast(walkstate, ss, walkparams, ipa,
```

```
                                                    descriptor);

            when DescriptorType_Invalid
                fault.statuscode = Fault_Translation;
                return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

            otherwise
                Unreachable();

    until desctype IN {DescriptorType_Page, DescriptorType_Block};

    if (walkstate.contiguous == '1' &&
            AArch64.ContiguousBitFaults(walkparams.txsz, walkparams.tgx, walkstate.level)) then
        fault.statuscode = Fault_Translation;
    elsif desctype == DescriptorType_Block && AArch64.BlocknTFaults(descriptor) then
        fault.statuscode = Fault_Translation;
    // Detect Address Size Fault by final output
    elsif AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, ipa_64) then
        fault.statuscode = Fault_AddressSize;
    // Check descriptor AF bit
    elsif (descriptor<10> == '0' && walkparams.ha == '0' &&
            !(acctype IN {AccType_DC, AccType_IC} &&
            !boolean IMPLEMENTATION_DEFINED "Generate access flag fault on IC/DC operations")) then
        fault.statuscode = Fault_AccessFlag;

    return (fault, walkaddress, walkstate, descriptor);
```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.SS2InitialTTWState

```
// AArch64.SS2InitialTTWState()
// ============================
// Set properties of first access to translation tables in Secure stage 2

TTWState AArch64.SS2InitialTTWState(S2TTWParams walkparams, PASpace ipaspace)
    TTWState    walkstate;
    FullAddress tablebase;

    bits(64) ttbr;
    if ipaspace == PAS_Secure then
        ttbr = VSTTBR_EL2;
    else
        ttbr = VTTBR_EL2;

    if ipaspace == PAS_Secure then
        if walkparams.sw == '0' then
            tablebase.paspace = PAS_Secure;
        else
            tablebase.paspace = PAS_NonSecure;
    else
        if walkparams.nsw == '0' then
            tablebase.paspace = PAS_Secure;
        else
            tablebase.paspace = PAS_NonSecure;

    startlevel        = AArch64.S2StartLevel(walkparams);
    tablebase.address = AArch64.TTBaseAddress(ttbr, walkparams.txsz, walkparams.ps, walkparams.ds,
                                              walkparams.tgx, startlevel);

    walkstate.baseaddress = tablebase;
    walkstate.level       = startlevel;
    walkstate.istable     = TRUE;
    walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);

    return walkstate;
```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.SS2OutputPASpace

```
// AArch64.SS2OutputPASpace()
// ==========================
// Assign PA Space to output of Secure stage 2 translation

PASpace AArch64.SS2OutputPASpace(S2TTWParams walkparams, PASpace ipaspace)
    if ipaspace == PAS_Secure then
        if walkparams.<sw,sa> == '00' then
            return PAS_Secure;
        else
            return PAS_NonSecure;
    else
        if walkparams.<sw,sa,nsw,nsa> == '0000' then
            return PAS_Secure;
        else
            return PAS_NonSecure;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.BBMSupportLevel

```
// AArch64.BBMSupportLevel()
// =========================
// Returns the level of FEAT_BBM supported

integer AArch64.BlockBBMSupportLevel()
    if !HaveBlockBBM() then
        return integer UNKNOWN;
    else
        return integer IMPLEMENTATION_DEFINED "Block BBM support level";
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.GetS1TTWParams

```
// AArch64.GetS1TTWParams()
// ========================
// Returns stage 1 translation table walk parameters from respective controlling
// System registers.

S1TTWParams AArch64.GetS1TTWParams(Regime regime, bits(64) va)
    S1TTWParams walkparams;

    varange = AArch64.GetVARange(va);

    case regime of
        when Regime_EL3  walkparams = AArch64.S1TTWParamsEL3();
        when Regime_EL2  walkparams = AArch64.S1TTWParamsEL2();
        when Regime_EL20 walkparams = AArch64.S1TTWParamsEL20(varange);
        when Regime_EL10 walkparams = AArch64.S1TTWParamsEL10(varange);

    maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
    mintxsz = AArch64.S1MinTxSZ(walkparams.ds, walkparams.tgx);
    if UInt(walkparams.txsz) > maxtxsz then
        if !(boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value above maximum") then
            walkparams.txsz = maxtxsz<5:0>;
    elsif !Have52BitVAExt() && UInt(walkparams.txsz) < mintxsz then
        if !(boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value below minimum") then
            walkparams.txsz = mintxsz<5:0>;

    return walkparams;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.GetS2TTWParams

```
// AArch64.GetS2TTWParams()
// ========================
// Gather walk parameters for stage 2 translation

S2TTWParams AArch64.GetS2TTWParams(SecurityState ss, PASpace ipaspace, boolean s1aarch64)
    S2TTWParams walkparams;

    if ss == SS_NonSecure then
        walkparams = AArch64.NSS2TTWParams(s1aarch64);
    elsif HaveSecureEL2Ext() && ss == SS_Secure then
        walkparams = AArch64.SS2TTWParams(ipaspace, s1aarch64);
    elsif ss == SS_Realm then
        // Realm stage 2 walk parameters are the same as for Non-secure
        walkparams = AArch64.NSS2TTWParams(s1aarch64);
    else
        Unreachable();

    maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
    mintxsz = AArch64.S2MinTxSZ(walkparams.ds, walkparams.tgx, s1aarch64);
    if UInt(walkparams.txsz) > maxtxsz then
        if !(boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value above maximum") then
            walkparams.txsz = maxtxsz<5:0>;
    elsif !Have52BitPAExt() && UInt(walkparams.txsz) < mintxsz then
        if !(boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value below minimum") then
            walkparams.txsz = mintxsz<5:0>;

    return walkparams;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.GetVARange

```
// AArch64.GetVARange()
// ====================
// Determines if the VA that is to be translated lies in LOWER or UPPER address range.

VARange AArch64.GetVARange(bits(64) va)
    if va<55> == '0' then
        return VARange_LOWER;
    else
        return VARange_UPPER;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.HaveS1TG

```
// AArch64.HaveS1TG()
// ==================
// Determine whether the given translation granule is supported for stage 1

boolean AArch64.HaveS1TG(TGx tgx)
    case tgx of
        when TGx_4KB  return boolean IMPLEMENTATION_DEFINED "Has 4K Translation Granule";
        when TGx_16KB return boolean IMPLEMENTATION_DEFINED "Has 16K Translation Granule";
        when TGx_64KB return boolean IMPLEMENTATION_DEFINED "Has 64K Translation Granule";
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.HaveS2TG

```
// AArch64.HaveS2TG()
// ==================
// Determine whether the given translation granule is supported for stage 2

boolean AArch64.HaveS2TG(TGx tgx)
    assert HaveEL(EL2);

    if HaveGTGExt() then
        case tgx of
            when TGx_4KB  return boolean IMPLEMENTATION_DEFINED "Has Stage 2 4K Translation Granule";
            when TGx_16KB return boolean IMPLEMENTATION_DEFINED "Has Stage 2 16K Translation Granule";
            when TGx_64KB return boolean IMPLEMENTATION_DEFINED "Has Stage 2 64K Translation Granule";
    else
        return AArch64.HaveS1TG(tgx);
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.MaxTxSZ

```
// AArch64.MaxTxSZ()
// =================
// Retrieve the maximum value of TxSZ indicating minimum input address size for both
// stages of translation

integer AArch64.MaxTxSZ(TGx tgx)
    if HaveSmallTranslationTableExt() then
        case tgx of
            when TGx_4KB   return 48;
            when TGx_16KB  return 48;
            when TGx_64KB  return 47;

    return 39;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.NSS2TTWParams

```
// AArch64.NSS2TTWParams()
// =======================
// Gather walk parameters specific for Non-secure stage 2 translation

S2TTWParams AArch64.NSS2TTWParams(boolean s1aarch64)
    S2TTWParams walkparams;

    walkparams.vm   = HCR_EL2.VM OR HCR_EL2.DC;
    walkparams.tgx  = AArch64.S2DecodeTG0(VTCR_EL2.TG0);
    walkparams.txsz = VTCR_EL2.T0SZ;
    walkparams.sl0  = VTCR_EL2.SL0;
    walkparams.ps   = VTCR_EL2.PS;
    walkparams.irgn = VTCR_EL2.IRGN0;
    walkparams.orgn = VTCR_EL2.ORGN0;
    walkparams.sh   = VTCR_EL2.SH0;
    walkparams.ee   = SCTLR_EL2.EE;

    walkparams.ptw = if HCR_EL2.TGE == '0'         then HCR_EL2.PTW else '0';
    walkparams.fwb = if HaveStage2MemAttrControl() then HCR_EL2.FWB else '0';
    walkparams.ha  = if HaveAccessFlagUpdateExt()  then VTCR_EL2.HA else '0';
    walkparams.hd  = if HaveDirtyBitModifierExt()  then VTCR_EL2.HD else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAndPASpaceExt() then
        walkparams.ds = VTCR_EL2.DS;
    else
        walkparams.ds = '0';
    if walkparams.tgx == TGx_4KB && Have52BitIPAAndPASpaceExt() then
        walkparams.sl2 = VTCR_EL2.SL2 AND VTCR_EL2.DS;
    else
        walkparams.sl2 = '0';
    walkparams.cmow = if HaveFeatCMOW() && IsHCRXEL2Enabled() then HCRX_EL2.CMOW else '0';

    return walkparams;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.PAMax

```
// AArch64.PAMax()
// ===============
// Returns the IMPLEMENTATION DEFINED maximum number of bits capable of representing
// physical address for this processor

integer AArch64.PAMax()
    return integer IMPLEMENTATION_DEFINED "Maximum Physical Address Size";
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1DCacheEnabled

```
// AArch64.S1DCacheEnabled()
// =========================
// Determine cacheability of stage 1 data accesses

boolean AArch64.S1DCacheEnabled(Regime regime)
    case regime of
        when Regime_EL3  return SCTLR_EL3.C == '1';
        when Regime_EL2  return SCTLR_EL2.C == '1';
        when Regime_EL20 return SCTLR_EL2.C == '1';
        when Regime_EL10 return SCTLR_EL1.C == '1';
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1DecodeTG0

```
// AArch64.S1DecodeTG0()
// =====================
// Decode stage 1 granule size configuration bits TG0

TGx AArch64.S1DecodeTG0(bits(2) tg0_in)
    bits(2) tg0 = tg0_in;
    TGx tgx;

    if tg0 == '11' then
        tg0 = bits(2) IMPLEMENTATION_DEFINED "TG0 encoded granule size";

    case tg0 of
        when '00'   tgx = TGx_4KB;
        when '01'   tgx = TGx_64KB;
        when '10'   tgx = TGx_16KB;

    if !AArch64.HaveS1TG(tgx) then
        case bits(2) IMPLEMENTATION_DEFINED "TG0 encoded granule size" of
            when '00'   tgx = TGx_4KB;
            when '01'   tgx = TGx_64KB;
            when '10'   tgx = TGx_16KB;

    return tgx;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1DecodeTG1

```
// AArch64.S1DecodeTG1()
// =====================
// Decode stage 1 granule size configuration bits TG1

TGx AArch64.S1DecodeTG1(bits(2) tg1_in)
    bits(2) tg1 = tg1_in;
    TGx tgx;

    if tg1 == '00' then
        tg1 = bits(2) IMPLEMENTATION_DEFINED "TG1 encoded granule size";

    case tg1 of
        when '10'   tgx = TGx_4KB;
        when '11'   tgx = TGx_64KB;
        when '01'   tgx = TGx_16KB;

    if !AArch64.HaveS1TG(tgx) then
        case bits(2) IMPLEMENTATION_DEFINED "TG1 encoded granule size" of
            when '10'   tgx = TGx_4KB;
            when '11'   tgx = TGx_64KB;
            when '01'   tgx = TGx_16KB;

    return tgx;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1EPD

```
// AArch64.S1EPD()
// ===============
// Determine whether stage 1 translation table walk is allowed for the VA range

bit AArch64.S1EPD(Regime regime, bits(64) va)
    assert HasUnprivileged(regime);
    varange = AArch64.GetVARange(va);

    case regime of
        when Regime_EL20 return if varange == VARange_LOWER then TCR_EL2.EPD0 else TCR_EL2.EPD1;
        when Regime_EL10 return if varange == VARange_LOWER then TCR_EL1.EPD0 else TCR_EL1.EPD1;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1Enabled

```
// AArch64.S1Enabled()
// ===================
// Determine if stage 1 is enabled for the access type for this translation regime

boolean AArch64.S1Enabled(Regime regime, AccType acctype)
    case regime of
        when Regime_EL3  return SCTLR_EL3.M == '1';
        when Regime_EL2  return SCTLR_EL2.M == '1';
        when Regime_EL20 return SCTLR_EL2.M == '1';
        when Regime_EL10 return (!EL2Enabled() || HCR_EL2.<DC,TGE> == '00') && SCTLR_EL1.M == '1';
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1ICacheEnabled

```
// AArch64.S1ICacheEnabled()
// =========================
// Determine cacheability of stage 1 instruction fetches

boolean AArch64.S1ICacheEnabled(Regime regime)
    case regime of
        when Regime_EL3  return SCTLR_EL3.I == '1';
        when Regime_EL2  return SCTLR_EL2.I == '1';
        when Regime_EL20 return SCTLR_EL2.I == '1';
        when Regime_EL10 return SCTLR_EL1.I == '1';
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1MinTxSZ

```
// AArch64.S1MinTxSZ()
// ===================
// Retrieve the minimum value of TxSZ indicating maximum input address size for stage 1

integer AArch64.S1MinTxSZ(bit ds, TGx tgx)
    if (Have52BitVAExt() && tgx == TGx_64KB) || ds == '1' then
        return 12;

    return 16;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1TTBR

```
// AArch64.S1TTBR()
// ================
// Identify stage 1 table base register for the acting translation regime

bits(64) AArch64.S1TTBR(Regime regime, bits(64) va)
    varange = AArch64.GetVARange(va);

    case regime of
        when Regime_EL3  return TTBR0_EL3;
        when Regime_EL2  return TTBR0_EL2;
        when Regime_EL20 return if varange == VARange_LOWER then TTBR0_EL2 else TTBR1_EL2;
        when Regime_EL10 return if varange == VARange_LOWER then TTBR0_EL1 else TTBR1_EL1;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1TTWParamsEL10

```
// AArch64.S1TTWParamsEL10()
// =========================
// Gather stage 1 translation table walk parameters for EL1&0 regime
// (with EL2 enabled or disabled)

S1TTWParams AArch64.S1TTWParamsEL10(VARange varange)
    S1TTWParams walkparams;

    if varange == VARange_LOWER then
        walkparams.tgx  = AArch64.S1DecodeTG0(TCR_EL1.TG0);
        walkparams.txsz = TCR_EL1.T0SZ;
        walkparams.irgn = TCR_EL1.IRGN0;
        walkparams.orgn = TCR_EL1.ORGN0;
        walkparams.sh   = TCR_EL1.SH0;
        walkparams.tbi  = TCR_EL1.TBI0;

        walkparams.nfd  = if HaveSVE() || HaveTME() then TCR_EL1.NFD0  else '0';
        walkparams.tbid = if HavePACExt()           then TCR_EL1.TBID0 else '0';
        walkparams.e0pd = if HaveE0PDExt()          then TCR_EL1.E0PD0 else '0';
        walkparams.hpd  = if AArch64.HaveHPDExt()   then TCR_EL1.HPD0  else '0';
    else
        walkparams.tgx  = AArch64.S1DecodeTG1(TCR_EL1.TG1);
        walkparams.txsz = TCR_EL1.T1SZ;
        walkparams.irgn = TCR_EL1.IRGN1;
        walkparams.orgn = TCR_EL1.ORGN1;
        walkparams.sh   = TCR_EL1.SH1;
        walkparams.tbi  = TCR_EL1.TBI1;

        walkparams.nfd  = if HaveSVE() || HaveTME() then TCR_EL1.NFD1  else '0';
        walkparams.tbid = if HavePACExt()           then TCR_EL1.TBID1 else '0';
        walkparams.e0pd = if HaveE0PDExt()          then TCR_EL1.E0PD1 else '0';
        walkparams.hpd  = if AArch64.HaveHPDExt()   then TCR_EL1.HPD1  else '0';

    walkparams.mair = MAIR_EL1;
    walkparams.wxn  = SCTLR_EL1.WXN;
    walkparams.ps   = TCR_EL1.IPS;
    walkparams.ee   = SCTLR_EL1.EE;
    walkparams.sif  = SCR_EL3.SIF;

    if EL2Enabled() then
        walkparams.dc  = HCR_EL2.DC;
        walkparams.dct = if HaveMTE2Ext() then HCR_EL2.DCT else '0';

    if HaveTrapLoadStoreMultipleDeviceExt() then
        walkparams.ntlsmd = SCTLR_EL1.nTLSMD;
    else
        walkparams.ntlsmd = '1';

    if EL2Enabled() then
        if HCR_EL2.<NV,NV1> == '01' then
            case ConstrainUnpredictable(Unpredictable_NVNV1) of
                when Constraint_NVNV1_00 walkparams.nv1 = '0';
                when Constraint_NVNV1_01 walkparams.nv1 = '1';
                when Constraint_NVNV1_11 walkparams.nv1 = '1';
        else
            walkparams.nv1 = HCR_EL2.NV1;
    else
        walkparams.nv1 = '0';

    walkparams.epan = if HavePAN3Ext()             then SCTLR_EL1.EPAN else '0';
    walkparams.cmow = if HaveFeatCMOW()            then SCTLR_EL1.CMOW else '0';
    walkparams.ha   = if HaveAccessFlagUpdateExt() then TCR_EL1.HA else '0';
    walkparams.hd   = if HaveDirtyBitModifierExt() then TCR_EL1.HD else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAndPASpaceExt() then
        walkparams.ds = TCR_EL1.DS;
    else
        walkparams.ds = '0';

    return walkparams;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1TTWParamsEL2

```
// AArch64.S1TTWParamsEL2()
// =======================
// Gather stage 1 translation table walk parameters for EL2 regime

S1TTWParams AArch64.S1TTWParamsEL2()
    S1TTWParams walkparams;

    walkparams.tgx  = AArch64.S1DecodeTG0(TCR_EL2.TG0);
    walkparams.txsz = TCR_EL2.T0SZ;
    walkparams.ps   = TCR_EL2.PS;
    walkparams.irgn = TCR_EL2.IRGN0;
    walkparams.orgn = TCR_EL2.ORGN0;
    walkparams.sh   = TCR_EL2.SH0;
    walkparams.tbi  = TCR_EL2.TBI;
    walkparams.mair = MAIR_EL2;
    walkparams.wxn  = SCTLR_EL2.WXN;
    walkparams.ee   = SCTLR_EL2.EE;
    walkparams.sif  = SCR_EL3.SIF;

    walkparams.tbid = if HavePACExt()              then TCR_EL2.TBID else '0';
    walkparams.hpd  = if AArch64.HaveHPDExt()      then TCR_EL2.HPD  else '0';
    walkparams.ha   = if HaveAccessFlagUpdateExt() then TCR_EL2.HA   else '0';
    walkparams.hd   = if HaveDirtyBitModifierExt() then TCR_EL2.HD   else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAndPASpaceExt() then
        walkparams.ds = TCR_EL2.DS;
    else
        walkparams.ds = '0';

    return walkparams;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1TTWParamsEL20

```
// AArch64.S1TTWParamsEL20()
// =========================
// Gather stage 1 translation table walk parameters for EL2&0 regime

S1TTWParams AArch64.S1TTWParamsEL20(VARange varange)
    S1TTWParams walkparams;

    if varange == VARange_LOWER then
        walkparams.tgx  = AArch64.S1DecodeTG0(TCR_EL2.TG0);
        walkparams.txsz = TCR_EL2.T0SZ;
        walkparams.irgn = TCR_EL2.IRGN0;
        walkparams.orgn = TCR_EL2.ORGN0;
        walkparams.sh   = TCR_EL2.SH0;
        walkparams.tbi  = TCR_EL2.TBI0;

        walkparams.nfd  = if HaveSVE() || HaveTME() then TCR_EL2.NFD0  else '0';
        walkparams.tbid = if HavePACExt()           then TCR_EL2.TBID0 else '0';
        walkparams.e0pd = if HaveE0PDExt()          then TCR_EL2.E0PD0 else '0';
        walkparams.hpd  = if AArch64.HaveHPDExt()   then TCR_EL2.HPD0  else '0';
    else
        walkparams.tgx  = AArch64.S1DecodeTG1(TCR_EL2.TG1);
        walkparams.txsz = TCR_EL2.T1SZ;
        walkparams.irgn = TCR_EL2.IRGN1;
        walkparams.orgn = TCR_EL2.ORGN1;
        walkparams.sh   = TCR_EL2.SH1;
        walkparams.tbi  = TCR_EL2.TBI1;

        walkparams.nfd  = if HaveSVE() || HaveTME() then TCR_EL2.NFD1  else '0';
        walkparams.tbid = if HavePACExt()           then TCR_EL2.TBID1 else '0';
        walkparams.e0pd = if HaveE0PDExt()          then TCR_EL2.E0PD1 else '0';
        walkparams.hpd  = if AArch64.HaveHPDExt()   then TCR_EL2.HPD1  else '0';

    walkparams.mair = MAIR_EL2;
    walkparams.wxn  = SCTLR_EL2.WXN;
    walkparams.ps   = TCR_EL2.IPS;
    walkparams.ee   = SCTLR_EL2.EE;
    walkparams.sif  = SCR_EL3.SIF;

    if HaveTrapLoadStoreMultipleDeviceExt() then
        walkparams.ntlsmd = SCTLR_EL2.nTLSMD;
    else
        walkparams.ntlsmd = '1';

    walkparams.epan = if HavePAN3Ext()             then SCTLR_EL2.EPAN else '0';
    walkparams.cmow = if HaveFeatCMOW()            then SCTLR_EL2.CMOW else '0';
    walkparams.ha   = if HaveAccessFlagUpdateExt() then TCR_EL2.HA else '0';
    walkparams.hd   = if HaveDirtyBitModifierExt() then TCR_EL2.HD else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAndPASpaceExt() then
        walkparams.ds = TCR_EL2.DS;
    else
        walkparams.ds = '0';

    return walkparams;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1TTWParamsEL3

```
// AArch64.S1TTWParamsEL3()
// =======================
// Gather stage 1 translation table walk parameters for EL3 regime

S1TTWParams AArch64.S1TTWParamsEL3()
    S1TTWParams walkparams;

    walkparams.tgx  = AArch64.S1DecodeTG0(TCR_EL3.TG0);
    walkparams.txsz = TCR_EL3.T0SZ;
    walkparams.ps   = TCR_EL3.PS;
    walkparams.irgn = TCR_EL3.IRGN0;
    walkparams.orgn = TCR_EL3.ORGN0;
    walkparams.sh   = TCR_EL3.SH0;
    walkparams.tbi  = TCR_EL3.TBI;
    walkparams.mair = MAIR_EL3;
    walkparams.wxn  = SCTLR_EL3.WXN;
    walkparams.ee   = SCTLR_EL3.EE;
    walkparams.sif  = SCR_EL3.SIF;

    walkparams.tbid = if HavePACExt()              then TCR_EL3.TBID else '0';
    walkparams.hpd  = if AArch64.HaveHPDExt()      then TCR_EL3.HPD  else '0';
    walkparams.ha   = if HaveAccessFlagUpdateExt() then TCR_EL3.HA   else '0';
    walkparams.hd   = if HaveDirtyBitModifierExt() then TCR_EL3.HD   else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAndPASpaceExt() then
        walkparams.ds = TCR_EL3.DS;
    else
        walkparams.ds = '0';

    return walkparams;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S2DecodeTG0

```
// AArch64.S2DecodeTG0()
// =====================
// Decode stage 2 granule size configuration bits TG0

TGx AArch64.S2DecodeTG0(bits(2) tg0_in)
    bits(2) tg0 = tg0_in;
    TGx tgx;

    if tg0 == '11' then
        tg0 = bits(2) IMPLEMENTATION_DEFINED "TG0 encoded granule size";

    case tg0 of
        when '00'   tgx = TGx_4KB;
        when '01'   tgx = TGx_64KB;
        when '10'   tgx = TGx_16KB;

    if !AArch64.HaveS2TG(tgx) then
        case bits(2) IMPLEMENTATION_DEFINED "TG0 encoded granule size" of
            when '00'   tgx = TGx_4KB;
            when '01'   tgx = TGx_64KB;
            when '10'   tgx = TGx_16KB;

    return tgx;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S2MinTxSZ

```
// AArch64.S2MinTxSZ()
// ===================
// Retrieve the minimum value of TxSZ indicating maximum input address size for stage 2

integer AArch64.S2MinTxSZ(bit ds, TGx tgx, boolean s1aarch64)
    ips = AArch64.PAMax();

    if Have52BitPAExt() && tgx != TGx_64KB && ds == '0' then
        ips = Min(48, AArch64.PAMax());

    min_txsz = 64 - ips;
    if !s1aarch64 then
        // EL1 is AArch32
        min_txsz = Min(min_txsz, 24);

    return min_txsz;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.SS2TTWParams

```
// AArch64.SS2TTWParams()
// ======================
// Gather walk parameters specific for secure stage 2 translation

S2TTWParams AArch64.SS2TTWParams(PASpace ipaspace, boolean s1aarch64)
    S2TTWParams walkparams;

    if ipaspace == PAS_Secure then
        walkparams.tgx  = AArch64.S2DecodeTG0(VSTCR_EL2.TG0);
        walkparams.txsz = VSTCR_EL2.T0SZ;
        walkparams.sl0  = VSTCR_EL2.SL0;
        if walkparams.tgx == TGx_4KB && Have52BitIPAAndPASpaceExt() then
            walkparams.sl2 = VSTCR_EL2.SL2 AND VTCR_EL2.DS;
        else
            walkparams.sl2 = '0';
    elsif ipaspace == PAS_NonSecure then
        walkparams.tgx  = AArch64.S2DecodeTG0(VTCR_EL2.TG0);
        walkparams.txsz = VTCR_EL2.T0SZ;
        walkparams.sl0  = VTCR_EL2.SL0;
        if walkparams.tgx == TGx_4KB && Have52BitIPAAndPASpaceExt() then
            walkparams.sl2 = VTCR_EL2.SL2 AND VTCR_EL2.DS;
        else
            walkparams.sl2 = '0';
    else
        Unreachable();

    walkparams.sw   = VSTCR_EL2.SW;
    walkparams.nsw  = VTCR_EL2.NSW;
    walkparams.sa   = VSTCR_EL2.SA;
    walkparams.nsa  = VTCR_EL2.NSA;
    walkparams.vm   = HCR_EL2.VM OR HCR_EL2.DC;
    walkparams.ps   = VTCR_EL2.PS;
    walkparams.irgn = VTCR_EL2.IRGN0;
    walkparams.orgn = VTCR_EL2.ORGN0;
    walkparams.sh   = VTCR_EL2.SH0;
    walkparams.ee   = SCTLR_EL2.EE;

    walkparams.ptw = if HCR_EL2.TGE == '0'         then HCR_EL2.PTW else '0';
    walkparams.fwb = if HaveStage2MemAttrControl() then HCR_EL2.FWB else '0';
    walkparams.ha  = if HaveAccessFlagUpdateExt()  then VTCR_EL2.HA else '0';
    walkparams.hd  = if HaveDirtyBitModifierExt()  then VTCR_EL2.HD else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAndPASpaceExt() then
        walkparams.ds = VTCR_EL2.DS;
    else
        walkparams.ds = '0';
    walkparams.cmow = if HaveFeatCMOW() && IsHCRXEL2Enabled() then HCRX_EL2.CMOW else '0';

    return walkparams;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.VAMax

```
// AArch64.VAMax()
// ===============
// Returns the IMPLEMENTATION DEFINED maximum number of bits capable of representing
// the virtual address for this processor

integer AArch64.VAMax()
    return integer IMPLEMENTATION_DEFINED "Maximum Virtual Address Size";
```

Library pseudocode for shared/debug/ClearStickyErrors/ClearStickyErrors

```
// ClearStickyErrors()
// ===================

ClearStickyErrors()
    EDSCR.TXU = '0';            // Clear TX underrun flag
    EDSCR.RXO = '0';            // Clear RX overrun flag

    if Halted() then            // in Debug state
        EDSCR.ITO = '0';        // Clear ITR overrun flag

    // If halted and the ITR is not empty then it is UNPREDICTABLE whether the EDSCR.ERR is cleared.
    // The UNPREDICTABLE behavior also affects the instructions in flight, but this is not described
    // in the pseudocode.
    if Halted() && EDSCR.ITE == '0' && ConstrainUnpredictableBool(Unpredictable_CLEARERRITEZERO) then
        return;
    EDSCR.ERR = '0';            // Clear cumulative error flag

    return;
```

Library pseudocode for shared/debug/DebugTarget/DebugTarget

```
// DebugTarget()
// =============
// Returns the debug exception target Exception level

bits(2) DebugTarget()
    ss = CurrentSecurityState();
    return DebugTargetFrom(ss);
```

Library pseudocode for shared/debug/DebugTarget/DebugTargetFrom

```
// DebugTargetFrom()
// =================

bits(2) DebugTargetFrom(SecurityState from_state)
    boolean route_to_el2;
    if HaveEL(EL2) && (from_state != SS_Secure ||
        (HaveSecureEL2Ext() && (!HaveEL(EL3) || SCR_EL3.EEL2 == '1'))) then
        if ELUsingAArch32(EL2) then
            route_to_el2 = (HDCR.TDE == '1' || HCR.TGE == '1');
        else
            route_to_el2 = (MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1');
    else
        route_to_el2 = FALSE;

    bits(2) target;
    if route_to_el2 then
        target = EL2;
    elsif HaveEL(EL3) && !HaveAArch64() && from_state == SS_Secure then
        target = EL3;
    else
        target = EL1;

    return target;
```

Library pseudocode for shared/debug/DoubleLockStatus/DoubleLockStatus

```
// DoubleLockStatus()
// ==================
// Returns the state of the OS Double Lock.
//    FALSE if OSDLR_EL1.DLK == 0 or DBGPRCR_EL1.CORENPDRQ == 1 or the PE is in Debug state.
//    TRUE if OSDLR_EL1.DLK == 1 and DBGPRCR_EL1.CORENPDRQ == 0 and the PE is in Non-debug state.

boolean DoubleLockStatus()
    if !HaveDoubleLock() then
        return FALSE;
    elsif ELUsingAArch32(EL1) then
        return DBGOSDLR.DLK == '1' && DBGPRCR.CORENPDRQ == '0' && !Halted();
    else
        return OSDLR_EL1.DLK == '1' && DBGPRCR_EL1.CORENPDRQ == '0' && !Halted();
```

Library pseudocode for shared/debug/OSLockStatus/OSLockStatus

```
// OSLockStatus()
// ==============
// Returns the state of the OS Lock.

boolean OSLockStatus()
    return (if ELUsingAArch32(EL1) then DBGOSLSR.OSLK else OSLSR_EL1.OSLK) == '1';
```

Library pseudocode for shared/debug/SoftwareLockStatus/Component

```
enumeration Component {
        Component_PMU,
        Component_Debug,
        Component_CTI
};
```

Library pseudocode for shared/debug/SoftwareLockStatus/GetAccessComponent

```
// Returns the accessed component.
Component GetAccessComponent();
```

Library pseudocode for shared/debug/SoftwareLockStatus/SoftwareLockStatus

```
// SoftwareLockStatus()
// ====================
// Returns the state of the Software Lock.

boolean SoftwareLockStatus()
    Component component = GetAccessComponent();
    if !HaveSoftwareLock(component) then
        return FALSE;
    case component of
        when Component_Debug
            return EDLSR.SLK == '1';
        when Component_PMU
            return PMLSR.SLK == '1';
        when Component_CTI
            return CTILSR.SLK == '1';
        otherwise
            Unreachable();
```

Library pseudocode for shared/debug/authentication/AccessState

```
// Returns the Security state of the access.
SecurityState AccessState();
```

Library pseudocode for shared/debug/authentication/AllowExternalDebugAccess

```
// AllowExternalDebugAccess()
// ==========================
// Returns TRUE if an external debug interface access to the External debug registers
// is allowed, FALSE otherwise.

boolean AllowExternalDebugAccess()
    // The access may also be subject to OS Lock, power-down, etc.
    return AllowExternalDebugAccess(AccessState());

// AllowExternalDebugAccess()
// ==========================
// Returns TRUE if an external debug interface access to the External debug registers
// is allowed for the given Security state, FALSE otherwise.

boolean AllowExternalDebugAccess(SecurityState access_state)
    // The access may also be subject to OS Lock, power-down, etc.
    if HaveRME() then
        case MDCR_EL3.<EDADE,EDAD> of
            when '00' return TRUE;
            when '01' return access_state IN {SS_Root, SS_Secure};
            when '10' return access_state IN {SS_Root, SS_Realm};
            when '11' return access_state == SS_Root;

    if HaveSecureExtDebugView() then
        if access_state == SS_Secure then return TRUE;
    else
        if !ExternalInvasiveDebugEnabled() then return FALSE;
        if ExternalSecureInvasiveDebugEnabled() then return TRUE;

    if HaveEL(EL3) then
        EDAD_bit = if ELUsingAArch32(EL3) then SDCR.EDAD else MDCR_EL3.EDAD;
        return EDAD_bit == '0';
    else
        return NonSecureOnlyImplementation();
```

Library pseudocode for shared/debug/authentication/AllowExternalPMUAccess

```
// AllowExternalPMUAccess()
// ========================
// Returns TRUE if an external debug interface access to the PMU registers is
// allowed, FALSE otherwise.

boolean AllowExternalPMUAccess()
    // The access may also be subject to OS Lock, power-down, etc.
    return AllowExternalPMUAccess(AccessState());

// AllowExternalPMUAccess()
// ========================
// Returns TRUE if an external debug interface access to the PMU registers is
// allowed for the given Security state, FALSE otherwise.

boolean AllowExternalPMUAccess(SecurityState access_state)
    // The access may also be subject to OS Lock, power-down, etc.
    if HaveRME() then
        case MDCR_EL3.<EPMADE,EPMAD> of
            when '00' return TRUE;
            when '01' return access_state IN {SS_Root, SS_Secure};
            when '10' return access_state IN {SS_Root, SS_Realm};
            when '11' return access_state == SS_Root;

    if HaveSecureExtDebugView() then
        if access_state == SS_Secure then return TRUE;
    else
        if !ExternalInvasiveDebugEnabled() then return FALSE;
        if ExternalSecureInvasiveDebugEnabled() then return TRUE;

    if HaveEL(EL3) then
        EPMAD_bit = if ELUsingAArch32(EL3) then SDCR.EPMAD else MDCR_EL3.EPMAD;
        return EPMAD_bit == '0';
    else
        return NonSecureOnlyImplementation();
```

Library pseudocode for shared/debug/authentication/AllowExternalTraceAccess

```
// AllowExternalTraceAccess()
// ==========================
// Returns TRUE if an external Trace access to the Trace registers is allowed, FALSE otherwise.

boolean AllowExternalTraceAccess()
    if !HaveTraceBufferExtension() then
        return TRUE;
    else
        return AllowExternalTraceAccess(AccessState());

// AllowExternalTraceAccess()
// ==========================
// Returns TRUE if an external Trace access to the Trace registers is allowed for the
// given Security state, FALSE otherwise.

boolean AllowExternalTraceAccess(SecurityState access_state)
    // The access may also be subject to OS lock, power-down, etc.
    if !HaveTraceBufferExtension() then return TRUE;
    assert HaveSecureExtDebugView();
    if HaveRME() then
        case MDCR_EL3.<ETADE,ETAD> of
            when '00' return TRUE;
            when '01' return access_state IN {SS_Root, SS_Secure};
            when '10' return access_state IN {SS_Root, SS_Realm};
            when '11' return access_state == SS_Root;

    if access_state == SS_Secure then return TRUE;
    if HaveEL(EL3) then
        // External Trace access is not supported for EL3 using AArch32
        assert !ELUsingAArch32(EL3);
        return MDCR_EL3.ETAD == '0';
    else
        return NonSecureOnlyImplementation();
```

Library pseudocode for shared/debug/authentication/Debug_authentication

```
signal DBGEN;
signal NIDEN;
signal SPIDEN;
signal SPNIDEN;
signal RLPIDEN;
signal RTPIDEN;
```

Library pseudocode for shared/debug/authentication/ExternalInvasiveDebugEnabled

```
// ExternalInvasiveDebugEnabled()
// ==============================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the DBGEN signal.

boolean ExternalInvasiveDebugEnabled()
    return DBGEN == HIGH;
```

Library pseudocode for shared/debug/authentication/ExternalNoninvasiveDebugAllowed

```
// ExternalNoninvasiveDebugAllowed()
// =================================
// Returns TRUE if Trace and PC Sample-based Profiling are allowed

boolean ExternalNoninvasiveDebugAllowed()
    return ExternalNoninvasiveDebugAllowed(PSTATE.EL);

// ExternalNoninvasiveDebugAllowed()
// =================================

boolean ExternalNoninvasiveDebugAllowed(bits(2) el)
    if !ExternalNoninvasiveDebugEnabled() then return FALSE;
    ss = SecurityStateAtEL(el);

    if ((ELUsingAArch32(EL3) || ELUsingAArch32(EL1)) && el == EL0 &&
        ss == SS_Secure && SDER.SUNIDEN == '1') then
        return TRUE;

    case ss of
        when SS_NonSecure return TRUE;
        when SS_Secure    return ExternalSecureNoninvasiveDebugEnabled();
        when SS_Realm     return ExternalRealmNoninvasiveDebugEnabled();
        when SS_Root      return ExternalRootNoninvasiveDebugEnabled();
```

Library pseudocode for shared/debug/authentication/ExternalNoninvasiveDebugEnabled

```
// ExternalNoninvasiveDebugEnabled()
// =================================
// This function returns TRUE if the FEAT_Debugv8p4 is implemented.
// Otherwise, this function is IMPLEMENTATION DEFINED, and, in the
// recommended interface, ExternalNoninvasiveDebugEnabled returns
// the state of the (DBGEN OR NIDEN) signal.

boolean ExternalNoninvasiveDebugEnabled()
    return !HaveNoninvasiveDebugAuth() || ExternalInvasiveDebugEnabled() || NIDEN == HIGH;
```

Library pseudocode for shared/debug/authentication/ExternalRealmInvasiveDebugEnabled

```
// ExternalRealmInvasiveDebugEnabled()
// ===================================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the
// (DBGEN AND RLPIDEN) signal.

boolean ExternalRealmInvasiveDebugEnabled()
    if !HaveRME() then return FALSE;
    return ExternalInvasiveDebugEnabled() && RLPIDEN == HIGH;
```

Library pseudocode for shared/debug/authentication/ExternalRealmNoninvasiveDebugEnabled

```
// ExternalRealmNoninvasiveDebugEnabled()
// ======================================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the
// (DBGEN AND RLPIDEN) signal.

boolean ExternalRealmNoninvasiveDebugEnabled()
    if !HaveRME() then return FALSE;
    return ExternalRealmInvasiveDebugEnabled();
```

Library pseudocode for shared/debug/authentication/ExternalRootInvasiveDebugEnabled

```
// ExternalRootInvasiveDebugEnabled()
// ==================================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the
// (DBGEN AND RLPIDEN AND SPIDEN AND RTPIDEN) signal.

boolean ExternalRootInvasiveDebugEnabled()
    if !HaveRME() then return FALSE;
    return (ExternalInvasiveDebugEnabled() &&
            ExternalSecureInvasiveDebugEnabled() &&
            ExternalRealmInvasiveDebugEnabled() &&
            RTPIDEN == HIGH);
```

Library pseudocode for shared/debug/authentication/ExternalRootNoninvasiveDebugEnabled

```
// ExternalRootNoninvasiveDebugEnabled()
// =====================================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the
// (DBGEN AND RLPIDEN AND SPIDEN AND RTPIDEN) signal.

boolean ExternalRootNoninvasiveDebugEnabled()
    if !HaveRME() then return FALSE;
    return ExternalRootInvasiveDebugEnabled();
```

Library pseudocode for shared/debug/authentication/ExternalSecureInvasiveDebugEnabled

```
// ExternalSecureInvasiveDebugEnabled()
// ====================================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGEN AND SPIDEN) signal.
// CoreSight allows asserting SPIDEN without also asserting DBGEN, but this is not recommended.

boolean ExternalSecureInvasiveDebugEnabled()
    if !HaveEL(EL3) && !SecureOnlyImplementation() then return FALSE;
    return ExternalInvasiveDebugEnabled() && SPIDEN == HIGH;
```

Library pseudocode for shared/debug/authentication/ExternalSecureNoninvasiveDebugEnabled

```
// ExternalSecureNoninvasiveDebugEnabled()
// =======================================
// This function returns the value of ExternalSecureInvasiveDebugEnabled() when FEAT_Debugv8p4
// is implemented. Otherwise, the definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGEN OR NIDEN) AND
// (SPIDEN OR SPNIDEN) signal.

boolean ExternalSecureNoninvasiveDebugEnabled()
    if !HaveEL(EL3) && !SecureOnlyImplementation() then return FALSE;
    if HaveNoninvasiveDebugAuth() then
        return ExternalNoninvasiveDebugEnabled() && (SPIDEN == HIGH || SPNIDEN == HIGH);
    else
        return ExternalSecureInvasiveDebugEnabled();
```

Library pseudocode for shared/debug/authentication/IsAccessSecure

```
// Returns TRUE when an access is Secure
boolean IsAccessSecure();
```

Library pseudocode for shared/debug/authentication/IsCorePowered

```
// Returns TRUE if the Core power domain is powered on, FALSE otherwise.
boolean IsCorePowered();
```

Library pseudocode for shared/debug/breakpoint/CheckValidStateMatch

```
// CheckValidStateMatch()
// ======================
// Checks for an invalid state match that will generate Constrained
// Unpredictable behaviour, otherwise returns Constraint_NONE.

(Constraint, bits(2), bit, bit, bits(2)) CheckValidStateMatch(bits(2) SSC_in, bit SSCE_in, bit HMC_in,
                                                              bits(2) PxC_in, boolean isbreakpnt)
    if !HaveRME() then assert SSCE_in == '0';
    boolean reserved = FALSE;
    bits(2) SSC = SSC_in;
    bit SSCE = SSCE_in;
    bit HMC = HMC_in;
    bits(2) PxC = PxC_in;

    // Values that are not allocated in any architecture version
    case HMC:SSCE:SSC:PxC of
        when '0 0 11 10' reserved = TRUE;
        when '1 0 00 x0' reserved = TRUE;
        when '1 0 01 10' reserved = TRUE;
        when '1 0 1x 10' reserved = TRUE;
        when 'x 1 xx xx' reserved = SSC != '01' || (HMC:PxC) IN {'000','110'};
        otherwise        reserved = FALSE;

    // Match 'Usr/Sys/Svc' valid only for AArch32 breakpoints
    if (!isbreakpnt || !HaveAArch32EL(EL1)) && HMC:PxC == '000' && SSC != '11' then
        reserved = TRUE;

    // Both EL3 and EL2 are not implemented
    if !HaveEL(EL3) && !HaveEL(EL2) && (HMC != '0' || SSC != '00') then
        reserved = TRUE;

    // EL3 is not implemented
    if !HaveEL(EL3) && SSC IN {'01','10'} && HMC:SSC:PxC != '10100' then
        reserved = TRUE;

    // EL3 using AArch64 only
    if (!HaveEL(EL3) || !HaveAArch64()) && HMC:SSC:PxC == '11000' then
        reserved = TRUE;

    // EL2 is not implemented
    if !HaveEL(EL2) && HMC:SSC:PxC == '11100' then
        reserved = TRUE;

    // Secure EL2 is not implemented
    if !HaveSecureEL2Ext() && (HMC:SSC:PxC)  IN {'01100','10100','x11x1'} then
        reserved = TRUE;

    if reserved then
        // If parameters are set to a reserved type, behaves as either disabled or a defined type
        Constraint c;
        (c, <HMC,SSC,SSCE,PxC>) = ConstrainUnpredictableBits(Unpredictable_RESBPWPCTRL, 6);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then
            return (c, bits(2) UNKNOWN, bit UNKNOWN, bit UNKNOWN, bits(2) UNKNOWN);
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    return (Constraint_NONE, SSC, SSCE, HMC, PxC);
```

Library pseudocode for shared/debug/breakpoint/NumBreakpointsImplemented

```
// NumBreakpointsImplemented()
// ===========================
// Returns the number of breakpoints implemented. This is indicated to software by
// DBGDIDR.BRPs in AArch32 state, and ID_AA64DFR0_EL1.BRPs in AArch64 state.

integer NumBreakpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of breakpoints";
```

Library pseudocode for shared/debug/breakpoint/NumContextAwareBreakpointsImplemented

```
// NumContextAwareBreakpointsImplemented()
// =======================================
// Returns the number of context-aware breakpoints implemented. This is indicated to software by
// DBGDIDR.CTX_CMPs in AArch32 state, and ID_AA64DFR0_EL1.CTX_CMPs in AArch64 state.

integer NumContextAwareBreakpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of context-aware breakpoints";
```

Library pseudocode for shared/debug/breakpoint/NumWatchpointsImplemented

```
// NumWatchpointsImplemented()
// ===========================
// Returns the number of watchpoints implemented. This is indicated to software by
// DBGDIDR.WRPs in AArch32 state, and ID_AA64DFR0_EL1.WRPs in AArch64 state.

integer NumWatchpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of watchpoints";
```

Library pseudocode for shared/debug/cti/CTI_SetEventLevel

```
// Set a Cross Trigger multi-cycle input event trigger to the specified level.
CTI_SetEventLevel(CrossTriggerIn id, signal level);
```

Library pseudocode for shared/debug/cti/CTI_SignalEvent

```
// Signal a discrete event on a Cross Trigger input event trigger.
CTI_SignalEvent(CrossTriggerIn id);
```

Library pseudocode for shared/debug/cti/CrossTrigger

```
enumeration CrossTriggerOut {CrossTriggerOut_DebugRequest, CrossTriggerOut_RestartRequest,
                             CrossTriggerOut_IRQ,          CrossTriggerOut_RSVD3,
                             CrossTriggerOut_TraceExtIn0,  CrossTriggerOut_TraceExtIn1,
                             CrossTriggerOut_TraceExtIn2,  CrossTriggerOut_TraceExtIn3};

enumeration CrossTriggerIn  {CrossTriggerIn_CrossHalt,     CrossTriggerIn_PMUOverflow,
                             CrossTriggerIn_RSVD2,         CrossTriggerIn_RSVD3,
                             CrossTriggerIn_TraceExtOut0,  CrossTriggerIn_TraceExtOut1,
                             CrossTriggerIn_TraceExtOut2,  CrossTriggerIn_TraceExtOut3};
```

Library pseudocode for shared/debug/dccanditr/CheckForDCCInterrupts

```
// CheckForDCCInterrupts()
// =======================

CheckForDCCInterrupts()
    commrx = (EDSCR.RXfull == '1');
    commtx = (EDSCR.TXfull == '0');

    // COMMRX and COMMTX support is optional and not recommended for new designs.
    // SetInterruptRequestLevel(InterruptID_COMMRX, if commrx then HIGH else LOW);
    // SetInterruptRequestLevel(InterruptID_COMMTX, if commtx then HIGH else LOW);

    // The value to be driven onto the common COMMIRQ signal.
    boolean commirq;
    if ELUsingAArch32(EL1) then
        commirq = ((commrx && DBGDCCINT.RX == '1') ||
                   (commtx && DBGDCCINT.TX == '1'));
    else
        commirq = ((commrx && MDCCINT_EL1.RX == '1') ||
                   (commtx && MDCCINT_EL1.TX == '1'));
    SetInterruptRequestLevel(InterruptID_COMMIRQ, if commirq then HIGH else LOW);

    return;
```

Library pseudocode for shared/debug/dccanditr/DBGDTRRX_EL0

```
// DBGDTRRX_EL0[] (external write)
// ===============================
// Called on writes to debug register 0x08C.

DBGDTRRX_EL0[boolean memory_mapped] = bits(32) value

    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "generate error response";
        return;

    if EDSCR.ERR == '1' then return;                    // Error flag set: ignore write

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then return;   // Software lock locked: ignore write

    if EDSCR.RXfull == '1' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0') then
        EDSCR.RXO = '1';  EDSCR.ERR = '1';              // Overrun condition: ignore write
        return;

    EDSCR.RXfull = '1';
    DTRRX = value;

    if Halted() && EDSCR.MA == '1' then
        EDSCR.ITE = '0';                            // See comments in EDITR[] (external write)
        if !UsingAArch32() then
            ExecuteA64(0xD5330501<31:0>);               // A64 "MRS X1,DBGDTRRX_EL0"
            ExecuteA64(0xB8004401<31:0>);               // A64 "STR W1,[X0],#4"
            X[1, 64] = bits(64) UNKNOWN;
        else
            ExecuteT32(0xEE10<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/);  // T32 "MRS R1,DBGDTRRXint"
            ExecuteT32(0xF840<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/);  // T32 "STR R1,[R0],#4"
            R[1] = bits(32) UNKNOWN;
        // If the store aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
        if EDSCR.ERR == '1' then
            EDSCR.RXfull = bit UNKNOWN;
            DBGDTRRX_EL0 = bits(64) UNKNOWN;
        else
            // "MRS X1,DBGDTRRX_EL0" calls DBGDTR_EL0[] (read) which clears RXfull.
            assert EDSCR.RXfull == '0';

        EDSCR.ITE = '1';                                // See comments in EDITR[] (external write)
    return;

// DBGDTRRX_EL0[] (external read)
// ==============================

bits(32) DBGDTRRX_EL0[boolean memory_mapped]
    return DTRRX;
```

Library pseudocode for shared/debug/dccanditr/DBGDTRTX_EL0

```
// DBGDTRTX_EL0[] (external read)
// ==============================
// Called on reads of debug register 0x080.

bits(32) DBGDTRTX_EL0[boolean memory_mapped]

    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "generate error response";
        return bits(32) UNKNOWN;

    underrun = EDSCR.TXfull == '0' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0');
    value = if underrun then bits(32) UNKNOWN else DTRTX;

    if EDSCR.ERR == '1' then return value;              // Error flag set: no side-effects

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then           // Software lock locked: no side-effects
        return value;

    if underrun then
        EDSCR.TXU = '1';  EDSCR.ERR = '1';              // Underrun condition: block side-effects
        return value;                                   // Return UNKNOWN

    EDSCR.TXfull = '0';
    if Halted() && EDSCR.MA == '1' then
        EDSCR.ITE = '0';                                // See comments in EDITR[] (external write)

        if !UsingAArch32() then
            ExecuteA64(0xB8404401<31:0>);               // A64 "LDR W1,[X0],#4"
        else
            ExecuteT32(0xF850<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/);      // T32 "LDR R1,[R0],#4"
        // If the load aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
        if EDSCR.ERR == '1' then
            EDSCR.TXfull = bit UNKNOWN;
            DBGDTRTX_EL0 = bits(64) UNKNOWN;
        else
            if !UsingAArch32() then
                ExecuteA64(0xD5130501<31:0>);           // A64 "MSR DBGDTRTX_EL0,X1"
            else
                ExecuteT32(0xEE00<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/);  // T32 "MSR DBGDTRTXint,R1"
            // "MSR DBGDTRTX_EL0,X1" calls DBGDTR_EL0[] (write) which sets TXfull.
            assert EDSCR.TXfull == '1';
        if !UsingAArch32() then
            X[1, 64] = bits(64) UNKNOWN;
        else
            R[1] = bits(32) UNKNOWN;
        EDSCR.ITE = '1';                                // See comments in EDITR[] (external write)

    return value;

// DBGDTRTX_EL0[] (external write)
// ===============================

DBGDTRTX_EL0[boolean memory_mapped] = bits(32) value
    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then return;   // Software lock locked: ignore write
    DTRTX = value;
    return;
```

Library pseudocode for shared/debug/dccanditr/DBGDTR_EL0

```
// DBGDTR_EL0[] (write)
// ===================
// System register writes to DBGDTR_EL0, DBGDTRTX_EL0 (AArch64) and DBGDTRTXint (AArch32)

DBGDTR_EL0[] = bits(N) value_in
    bits(N) value = value_in;
    // For MSR DBGDTRTX_EL0,<Rt>  N=32, value=X[t]<31:0>, X[t]<63:32> is ignored
    // For MSR DBGDTR_EL0,<Xt>    N=64, value=X[t]<63:0>
    assert N IN {32,64};
    if EDSCR.TXfull == '1' then
        value = bits(N) UNKNOWN;
    // On a 64-bit write, implement a half-duplex channel
    if N == 64 then DTRRX = value<63:32>;
    DTRTX = value<31:0>;        // 32-bit or 64-bit write
    EDSCR.TXfull = '1';
    return;

// DBGDTR_EL0[] (read)
// ==================
// System register reads of DBGDTR_EL0, DBGDTRRX_EL0 (AArch64) and DBGDTRRXint (AArch32)

bits(N) DBGDTR_EL0[]
    // For MRS <Rt>,DBGDTRTX_EL0  N=32, X[t]=Zeros(32):result
    // For MRS <Xt>,DBGDTR_EL0    N=64, X[t]=result
    assert N IN {32,64};
    bits(N) result;
    if EDSCR.RXfull == '0' then
        result = bits(N) UNKNOWN;
    else
        // On a 64-bit read, implement a half-duplex channel
        // NOTE: the word order is reversed on reads with regards to writes
        if N == 64 then result<63:32> = DTRTX;
        result<31:0> = DTRRX;
    EDSCR.RXfull = '0';
    return result;
```

Library pseudocode for shared/debug/dccanditr/DTR

```
bits(32) DTRRX;
bits(32) DTRTX;
```

Library pseudocode for shared/debug/dccanditr/EDITR

```
// EDITR[] (external write)
// ========================
// Called on writes to debug register 0x084.

EDITR[boolean memory_mapped] = bits(32) value
    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "generate error response";
        return;

    if EDSCR.ERR == '1' then return;                    // Error flag set: ignore write

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then return;   // Software lock locked: ignore write

    if !Halted() then return;                           // Non-debug state: ignore write

    if EDSCR.ITE == '0' || EDSCR.MA == '1' then
        EDSCR.ITO = '1';  EDSCR.ERR = '1';              // Overrun condition: block write
        return;

    // ITE indicates whether the processor is ready to accept another instruction; the processor
    // may support multiple outstanding instructions. Unlike the "InstrCompl" flag in [v7A] there
    // is no indication that the pipeline is empty (all instructions have completed). In this
    // pseudocode, the assumption is that only one instruction can be executed at a time,
    // meaning ITE acts like "InstrCompl".
    EDSCR.ITE = '0';

    if !UsingAArch32() then
        ExecuteA64(value);
    else
        ExecuteT32(value<15:0>/*hw1*/, value<31:16> /*hw2*/);

    EDSCR.ITE = '1';

    return;
```

Library pseudocode for shared/debug/halting/DCPSInstruction

```
// DCPSInstruction()
// =================
// Operation of the DCPS instruction in Debug state

DCPSInstruction(bits(2) target_el)

    SynchronizeContext();

    bits(2) handle_el;
    case target_el of
        when EL1
            if PSTATE.EL == EL2 || (PSTATE.EL == EL3 && !UsingAArch32()) then
                handle_el = PSTATE.EL;
            elsif EL2Enabled() && HCR_EL2.TGE == '1' then
                UNDEFINED;
            else
                handle_el = EL1;
        when EL2
            if !HaveEL(EL2) then
                UNDEFINED;
            elsif PSTATE.EL == EL3 && !UsingAArch32() then
                handle_el = EL3;
            elsif !IsSecureEL2Enabled() && CurrentSecurityState() == SS_Secure then
                UNDEFINED;
            else
                handle_el = EL2;
        when EL3
            if EDSCR.SDD == '1' || !HaveEL(EL3) then
                UNDEFINED;
            else
                handle_el = EL3;
        otherwise
            Unreachable();

    from_secure = CurrentSecurityState() == SS_Secure;
    if ELUsingAArch32(handle_el) then
        if PSTATE.M == M32_Monitor then SCR.NS = '0';
        assert UsingAArch32();                  // Cannot move from AArch64 to AArch32
        case handle_el of
            when EL1
                AArch32.WriteMode(M32_Svc);
                if HavePANExt() && SCTLR.SPAN == '0' then
                    PSTATE.PAN = '1';
            when EL2  AArch32.WriteMode(M32_Hyp);
            when EL3
                AArch32.WriteMode(M32_Monitor);
                if HavePANExt() then
                    if !from_secure then
                        PSTATE.PAN = '0';
                    elsif SCTLR.SPAN == '0' then
                        PSTATE.PAN = '1';
        if handle_el == EL2 then
            ELR_hyp = bits(32) UNKNOWN;  HSR = bits(32) UNKNOWN;
        else
            LR = bits(32) UNKNOWN;
        SPSR[] = bits(32) UNKNOWN;
        PSTATE.E = SCTLR[].EE;
        DLR = bits(32) UNKNOWN;  DSPSR = bits(32) UNKNOWN;

    else                                        // Targeting AArch64
        from_32 = UsingAArch32();
        if from_32 then AArch64.MaybeZeroRegisterUppers();
        if from_32 && HaveSME() && PSTATE.SM == '1' then
            ResetSVEState();
        else
            MaybeZeroSVEUppers(target_el);
        PSTATE.nRW = '0';  PSTATE.SP = '1';  PSTATE.EL = handle_el;
        if HavePANExt() && ((handle_el == EL1 && SCTLR_EL1.SPAN == '0') ||
                            (handle_el == EL2 && HCR_EL2.E2H == '1' &&
                             HCR_EL2.TGE == '1' && SCTLR_EL2.SPAN == '0')) then
```

```
            PSTATE.PAN = '1';
        ELR[] = bits(64) UNKNOWN;  SPSR[] = bits(64) UNKNOWN;  ESR[] = bits(64) UNKNOWN;
        DLR_EL0 = bits(64) UNKNOWN;  DSPSR_EL0 = bits(64) UNKNOWN;
        if HaveUAOExt() then PSTATE.UAO = '0';
        if HaveMTEExt() then PSTATE.TCO = '1';

    UpdateEDSCRFields();                          // Update EDSCR PE state flags
    sync_errors = HaveIESB() && SCTLR[].IESB == '1';
    if HaveDoubleFaultExt() && !UsingAArch32() then
        sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3);
    // SCTLR[].IESB might be ignored in Debug state.
    if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
        sync_errors = FALSE;
    if sync_errors then
        SynchronizeErrors();
    return;
```

Library pseudocode for shared/debug/halting/DRPSInstruction

```
// DRPSInstruction()
// =================
// Operation of the A64 DRPS and T32 ERET instructions in Debug state

DRPSInstruction()

    SynchronizeContext();

    sync_errors = HaveIESB() && SCTLR[].IESB == '1';
    if HaveDoubleFaultExt() && !UsingAArch32() then
        sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3);
    // SCTLR[].IESB might be ignored in Debug state.
    if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
        sync_errors = FALSE;
    if sync_errors then
        SynchronizeErrors();

    DebugRestorePSR();

    return;
```

Library pseudocode for shared/debug/halting/DebugHalt

```
constant bits(6) DebugHalt_Breakpoint      = '000111';
constant bits(6) DebugHalt_EDBGRQ          = '010011';
constant bits(6) DebugHalt_Step_Normal     = '011011';
constant bits(6) DebugHalt_Step_Exclusive  = '011111';
constant bits(6) DebugHalt_OSUnlockCatch   = '100011';
constant bits(6) DebugHalt_ResetCatch      = '100111';
constant bits(6) DebugHalt_Watchpoint      = '101011';
constant bits(6) DebugHalt_HaltInstruction = '101111';
constant bits(6) DebugHalt_SoftwareAccess  = '110011';
constant bits(6) DebugHalt_ExceptionCatch  = '110111';
constant bits(6) DebugHalt_Step_NoSyndrome = '111011';
```

Library pseudocode for shared/debug/halting/DebugRestorePSR

```
// DebugRestorePSR()
// =================

DebugRestorePSR()
    // PSTATE.{N,Z,C,V,Q,GE,SS,D,A,I,F} are not observable and ignored in Debug state, so
    // behave as if UNKNOWN.
    if UsingAArch32() then
        bits(32) spsr = SPSR[];
        SetPSTATEFromPSR(spsr);
        PSTATE.<N,Z,C,V,Q,GE,SS,A,I,F> = bits(13) UNKNOWN;
        //  In AArch32, all instructions are T32 and unconditional.
        PSTATE.IT = '00000000';  PSTATE.T = '1';        // PSTATE.J is RES0
        DLR = bits(32) UNKNOWN;  DSPSR = bits(32) UNKNOWN;
    else
        bits(64) spsr = SPSR[];
        SetPSTATEFromPSR(spsr);
        PSTATE.<N,Z,C,V,SS,D,A,I,F> = bits(9) UNKNOWN;
        DLR_EL0 = bits(64) UNKNOWN;  DSPSR_EL0 = bits(64) UNKNOWN;
    UpdateEDSCRFields();                                // Update EDSCR PE state flags
```

Library pseudocode for shared/debug/halting/DisableITRAndResumeInstructionPrefetch

```
DisableITRAndResumeInstructionPrefetch();
```

Library pseudocode for shared/debug/halting/ExecuteA64

```
// Execute an A64 instruction in Debug state.
ExecuteA64(bits(32) instr);
```

Library pseudocode for shared/debug/halting/ExecuteT32

```
// Execute a T32 instruction in Debug state.
ExecuteT32(bits(16) hw1, bits(16) hw2);
```

Library pseudocode for shared/debug/halting/ExitDebugState

```
// ExitDebugState()
// ================

ExitDebugState()
    assert Halted();
    SynchronizeContext();

    // Although EDSCR.STATUS signals that the PE is restarting, debuggers must use EDPRSR.SDR to
    // detect that the PE has restarted.
    EDSCR.STATUS = '000001';                            // Signal restarting
    // Clear any pending Halting debug events
    if Havev8p8Debug() then
        EDESR<3:0> = '0000';
    else
        EDESR<2:0> = '000';

    bits(64) new_pc;
    bits(64) spsr;

    if UsingAArch32() then
        new_pc = ZeroExtend(DLR, 64);
        spsr = ZeroExtend(DSPSR, 64);
    else
        new_pc = DLR_EL0;
        spsr = DSPSR_EL0;
    boolean illegal_psr_state = IllegalExceptionReturn(spsr);
    // If this is an illegal return, SetPSTATEFromPSR() will set PSTATE.IL.
    if UsingAArch32() then
        SetPSTATEFromPSR(spsr<31:0>);                   // Can update privileged bits, even at EL0
    else
        SetPSTATEFromPSR(spsr);                         // Can update privileged bits, even at EL0

    boolean branch_conditional = FALSE;
    if UsingAArch32() then
        if ConstrainUnpredictableBool(Unpredictable_RESTARTALIGNPC) then new_pc<0> = '0';
        // AArch32 branch
        BranchTo(new_pc<31:0>, BranchType_DBGEXIT, branch_conditional);
    else
        // If targeting AArch32 then PC[63:32,1:0] might be set to UNKNOWN.
        if illegal_psr_state && spsr<4> == '1' then
            new_pc<63:32> = bits(32) UNKNOWN;
            new_pc<1:0> = bits(2) UNKNOWN;
        if HaveBRBExt() then
            BRBEDebugStateExit(new_pc);
        // A type of branch that is never predicted
        BranchTo(new_pc, BranchType_DBGEXIT, branch_conditional);

    (EDSCR.STATUS,EDPRSR.SDR) = ('000010','1');         // Atomically signal restarted
    UpdateEDSCRFields();                                // Stop signalling PE state
    DisableITRAndResumeInstructionPrefetch();

    return;
```

Library pseudocode for shared/debug/halting/Halt

```
// Halt()
// ======

Halt(bits(6) reason)
    boolean is_async = FALSE;
    Halt(reason, is_async);

// Halt()
// ======

Halt(bits(6) reason, boolean is_async)

    CTI_SignalEvent(CrossTriggerIn_CrossHalt);  // Trigger other cores to halt

    bits(64) preferred_restart_address = ThisInstrAddr(64);
    bits(32) spsr_32;
    bits(64) spsr_64;
    if UsingAArch32() then
        spsr_32 = GetPSRFromPSTATE(DebugState, 32);
    else
        spsr_64 = GetPSRFromPSTATE(DebugState, 64);

    if (HaveBTIExt() && !is_async && !(reason IN {DebugHalt_Step_Normal, DebugHalt_Step_Exclusive,
        DebugHalt_Step_NoSyndrome, DebugHalt_Breakpoint, DebugHalt_HaltInstruction}) &&
        ConstrainUnpredictableBool(Unpredictable_ZEROBTYPE)) then
        if UsingAArch32() then
            spsr_32<11:10> = '00';
        else
            spsr_64<11:10> = '00';

    if UsingAArch32() then
        DLR = preferred_restart_address<31:0>;
        DSPSR = spsr_32;
    else
        DLR_EL0 = preferred_restart_address;
        DSPSR_EL0 = spsr_64;

    EDSCR.ITE = '1';
    EDSCR.ITO = '0';
    if HaveRME() then
        EDSCR.SDD = if ExternalRootInvasiveDebugEnabled() then '0' else '1';
    elsif CurrentSecurityState() == SS_Secure then
        EDSCR.SDD = '0';                        // If entered in Secure state, allow debug
    elsif HaveEL(EL3) then
        EDSCR.SDD = if ExternalSecureInvasiveDebugEnabled() then '0' else '1';
    else
        EDSCR.SDD = '1';                        // Otherwise EDSCR.SDD is RES1
    EDSCR.MA = '0';

    // In Debug state:
    // * PSTATE.{SS,SSBS,D,A,I,F} are not observable and ignored so behave-as-if UNKNOWN.
    // * PSTATE.{N,Z,C,V,Q,GE,E,M,nRW,EL,SP,DIT} are also not observable, but since these
    //     are not changed on exception entry, this function also leaves them unchanged.
    // * PSTATE.{IT,T} are ignored.
    // * PSTATE.IL is ignored and behave-as-if 0.
    // * PSTATE.BTYPE is ignored and behave-as-if 0.
    // * PSTATE.TCO is set 1.
    // * PSTATE.{UAO,PAN} are observable and not changed on entry into Debug state.
    if UsingAArch32() then
        PSTATE.<IT,SS,SSBS,A,I,F,T> = bits(14) UNKNOWN;
    else
        PSTATE.<SS,SSBS,D,A,I,F>    = bits(6)  UNKNOWN;
        PSTATE.TCO = '1';
        PSTATE.BTYPE = '00';
    PSTATE.IL = '0';
    StopInstructionPrefetchAndEnableITR();
    EDSCR.STATUS = reason;                      // Signal entered Debug state
    UpdateEDSCRFields();                        // Update EDSCR PE state flags.
    return;
```

Library pseudocode for shared/debug/halting/HaltOnBreakpointOrWatchpoint

```
// HaltOnBreakpointOrWatchpoint()
// ==============================
// Returns TRUE if the Breakpoint and Watchpoint debug events should be considered for Debug
// state entry, FALSE if they should be considered for a debug exception.

boolean HaltOnBreakpointOrWatchpoint()
    return HaltingAllowed() && EDSCR.HDE == '1' && OSLSR_EL1.OSLK == '0';
```

Library pseudocode for shared/debug/halting/Halted

```
// Halted()
// ========

boolean Halted()
    return !(EDSCR.STATUS IN {'000001', '000010'});                     // Halted
```

Library pseudocode for shared/debug/halting/HaltingAllowed

```
// HaltingAllowed()
// ================
// Returns TRUE if halting is currently allowed, FALSE if halting is prohibited.

boolean HaltingAllowed()
    if Halted() || DoubleLockStatus() then
        return FALSE;
    ss = CurrentSecurityState();
    case ss of
        when SS_NonSecure return ExternalInvasiveDebugEnabled();
        when SS_Secure    return ExternalSecureInvasiveDebugEnabled();
        when SS_Root      return ExternalRootInvasiveDebugEnabled();
        when SS_Realm     return ExternalRealmInvasiveDebugEnabled();
```

Library pseudocode for shared/debug/halting/Restarting

```
// Restarting()
// ============

boolean Restarting()
    return EDSCR.STATUS == '000001';                                    // Restarting
```

Library pseudocode for shared/debug/halting/StopInstructionPrefetchAndEnableITR

```
StopInstructionPrefetchAndEnableITR();
```

Library pseudocode for shared/debug/halting/UpdateEDSCRFields

```
// UpdateEDSCRFields()
// ===================
// Update EDSCR PE state fields

UpdateEDSCRFields()

    if !Halted() then
        EDSCR.EL = '00';
        if HaveRME() then
            EDSCR.<NSE,NS> = bits(2) UNKNOWN;
        else
            EDSCR.NS = bit UNKNOWN;

        EDSCR.RW = '1111';
    else
        EDSCR.EL = PSTATE.EL;
        ss = CurrentSecurityState();
        if HaveRME() then
            case ss of
                when SS_Secure    EDSCR.<NSE,NS> = '00';
                when SS_NonSecure EDSCR.<NSE,NS> = '01';
                when SS_Root      EDSCR.<NSE,NS> = '10';
                when SS_Realm     EDSCR.<NSE,NS> = '11';
        else
            EDSCR.NS = if ss == SS_Secure then '0' else '1';

        bits(4) RW;
        RW<1> = if ELUsingAArch32(EL1) then '0' else '1';
        if PSTATE.EL != EL0 then
            RW<0> = RW<1>;
        else
            RW<0> = if UsingAArch32() then '0' else '1';
        if !HaveEL(EL2) || (HaveEL(EL3) && SCR_GEN[].NS == '0' && !IsSecureEL2Enabled()) then
            RW<2> = RW<1>;
        else
            RW<2> = if ELUsingAArch32(EL2) then '0' else '1';
        if !HaveEL(EL3) then
            RW<3> = RW<2>;
        else
            RW<3> = if ELUsingAArch32(EL3) then '0' else '1';

        // The least-significant bits of EDSCR.RW are UNKNOWN if any higher EL is using AArch32.
        if RW<3> == '0' then RW<2:0> = bits(3) UNKNOWN;
        elsif RW<2> == '0' then RW<1:0> = bits(2) UNKNOWN;
        elsif RW<1> == '0' then RW<0> = bit UNKNOWN;
        EDSCR.RW = RW;
    return;
```

Library pseudocode for shared/debug/haltingevents/CheckExceptionCatch

```
// CheckExceptionCatch()
// =====================
// Check whether an Exception Catch debug event is set on the current Exception level

CheckExceptionCatch(boolean exception_entry)
    // Called after an exception entry or exit, that is, such that the Security state
    // and PSTATE.EL are correct for the exception target. When FEAT_Debugv8p2
    // is not implemented, this function might also be called at any time.
    ss = SecurityStateAtEL(PSTATE.EL);
    integer base;

    case ss of
        when SS_Secure    base = 0;
        when SS_NonSecure base = 4;
        when SS_Realm     base = 16;
        when SS_Root      base = 0;
    if HaltingAllowed() then
        boolean halt;
        if HaveExtendedECDebugEvents() then
            exception_exit = !exception_entry;
            increment = if ss == SS_Realm then 4 else 8;
            ctrl = EDECCR<UInt(PSTATE.EL) + base + increment>:EDECCR<UInt(PSTATE.EL) + base>;
            case ctrl of
                when '00'  halt = FALSE;
                when '01'  halt = TRUE;
                when '10'  halt = (exception_exit == TRUE);
                when '11'  halt = (exception_entry == TRUE);
        else
            halt = (EDECCR<UInt(PSTATE.EL) + base> == '1');

        if halt then
            if Havev8p8Debug() && exception_entry then
                EDESR.EC = '1';
            else
                Halt(DebugHalt_ExceptionCatch);
```

Library pseudocode for shared/debug/haltingevents/CheckHaltingStep

```
// CheckHaltingStep()
// ==================
// Check whether EDESR.SS has been set by Halting Step

CheckHaltingStep(boolean is_async)
    if HaltingAllowed() && EDESR.SS == '1' then
        // The STATUS code depends on how we arrived at the state where EDESR.SS == 1.
        if HaltingStep_DidNotStep() then
            Halt(DebugHalt_Step_NoSyndrome, is_async);
        elsif HaltingStep_SteppedEX() then
            Halt(DebugHalt_Step_Exclusive, is_async);
        else
            Halt(DebugHalt_Step_Normal, is_async);
```

Library pseudocode for shared/debug/haltingevents/CheckOSUnlockCatch

```
// CheckOSUnlockCatch()
// ====================
// Called on unlocking the OS Lock to pend an OS Unlock Catch debug event

CheckOSUnlockCatch()
    if ((HaveDoPD() && CTIDEVCTL.OSUCE == '1') ||
        (!HaveDoPD() && EDECR.OSUCE == '1')) then
        if !Halted() then EDESR.OSUC = '1';
```

Library pseudocode for shared/debug/haltingevents/CheckPendingExceptionCatch

```
// CheckPendingExceptionCatch()
// ============================
// Check whether EDESR.EC has been set by an Exception Catch debug event.

CheckPendingExceptionCatch(boolean is_async)
    if Havev8p8Debug() && HaltingAllowed() && EDESR.EC == '1' then
        Halt(DebugHalt_ExceptionCatch, is_async);
```

Library pseudocode for shared/debug/haltingevents/CheckPendingOSUnlockCatch

```
// CheckPendingOSUnlockCatch()
// ===========================
// Check whether EDESR.OSUC has been set by an OS Unlock Catch debug event

CheckPendingOSUnlockCatch()
    if HaltingAllowed() && EDESR.OSUC == '1' then
        boolean is_async = TRUE;
        Halt(DebugHalt_OSUnlockCatch, is_async);
```

Library pseudocode for shared/debug/haltingevents/CheckPendingResetCatch

```
// CheckPendingResetCatch()
// ========================
// Check whether EDESR.RC has been set by a Reset Catch debug event

CheckPendingResetCatch()
    if HaltingAllowed() && EDESR.RC == '1' then
        boolean is_async = TRUE;
        Halt(DebugHalt_ResetCatch, is_async);
```

Library pseudocode for shared/debug/haltingevents/CheckResetCatch

```
// CheckResetCatch()
// =================
// Called after reset

CheckResetCatch()
    if (HaveDoPD() && CTIDEVCTL.RCE == '1') || (!HaveDoPD() && EDECR.RCE == '1') then
        EDESR.RC = '1';
        // If halting is allowed then halt immediately
        if HaltingAllowed() then Halt(DebugHalt_ResetCatch);
```

Library pseudocode for shared/debug/haltingevents/CheckSoftwareAccessToDebugRegisters

```
// CheckSoftwareAccessToDebugRegisters()
// =====================================
// Check for access to Breakpoint and Watchpoint registers.

CheckSoftwareAccessToDebugRegisters()
    os_lock = (if ELUsingAArch32(EL1) then DBGOSLSR.OSLK else OSLSR_EL1.OSLK);
    if HaltingAllowed() && EDSCR.TDA == '1' && os_lock == '0' then
        Halt(DebugHalt_SoftwareAccess);
```

Library pseudocode for shared/debug/haltingevents/ExternalDebugRequest

```
// ExternalDebugRequest()
// ======================

ExternalDebugRequest()
    if HaltingAllowed() then
        boolean is_async = TRUE;
        Halt(DebugHalt_EDBGRQ, is_async);
    // Otherwise the CTI continues to assert the debug request until it is taken.
```

Library pseudocode for shared/debug/haltingevents/HaltingStep_DidNotStep

```
// Returns TRUE if the previously executed instruction was executed in the inactive state, that is,
// if it was not itself stepped.
boolean HaltingStep_DidNotStep();
```

Library pseudocode for shared/debug/haltingevents/HaltingStep_SteppedEX

```
// Returns TRUE if the previously executed instruction was a Load-Exclusive class instruction
// executed in the active-not-pending state.
boolean HaltingStep_SteppedEX();
```

Library pseudocode for shared/debug/haltingevents/RunHaltingStep

```
// RunHaltingStep()
// ================

RunHaltingStep(boolean exception_generated, bits(2) exception_target, boolean syscall,
               boolean reset)
    // "exception_generated" is TRUE if the previous instruction generated a synchronous exception
    // or was cancelled by an asynchronous exception.
    //
    // if "exception_generated" is TRUE then "exception_target" is the target of the exception, and
    // "syscall" is TRUE if the exception is a synchronous exception where the preferred return
    // address is the instruction following that which generated the exception.
    //
    // "reset" is TRUE if exiting reset state into the highest EL.

    if reset then assert !Halted();             // Cannot come out of reset halted
    active = EDECR.SS == '1' && !Halted();

    if active && reset then                     // Coming out of reset with EDECR.SS set
        EDESR.SS = '1';
    elsif active && HaltingAllowed() then
        boolean advance;
        if exception_generated && exception_target == EL3 then
            advance = syscall || ExternalSecureInvasiveDebugEnabled();
        else
            advance = TRUE;
        if advance then EDESR.SS = '1';

    return;
```

Library pseudocode for shared/debug/interrupts/ExternalDebugInterruptsDisabled

```
// ExternalDebugInterruptsDisabled()
// =================================
// Determine whether EDSCR disables interrupts routed to 'target'.

boolean ExternalDebugInterruptsDisabled(bits(2) target)
    boolean int_dis;
    SecurityState ss = SecurityStateAtEL(target);
    if Havev8p4Debug() then
        if EDSCR.INTdis[0] == '1' then
            case ss of
                when SS_NonSecure int_dis = ExternalInvasiveDebugEnabled();
                when SS_Secure    int_dis = ExternalSecureInvasiveDebugEnabled();
                when SS_Realm     int_dis = ExternalRealmInvasiveDebugEnabled();
                when SS_Root      int_dis = ExternalRootInvasiveDebugEnabled();
        else
            int_dis = FALSE;
    else
        case target of
            when EL3
                int_dis = (EDSCR.INTdis == '11' && ExternalSecureInvasiveDebugEnabled());
            when EL2
                int_dis = (EDSCR.INTdis IN {'1x'} && ExternalInvasiveDebugEnabled());
            when EL1
                if ss == SS_Secure then
                    int_dis = (EDSCR.INTdis IN {'1x'} && ExternalSecureInvasiveDebugEnabled());
                else
                    int_dis = (EDSCR.INTdis != '00' && ExternalInvasiveDebugEnabled());
    return int_dis;
```

Library pseudocode for shared/debug/pmu/GetNumEventCounters

```
// GetNumEventCounters()
// =====================
// Returns the number of event counters implemented. This is indicated to software at the
// highest Exception level by PMCR.N in AArch32 state, and PMCR_EL0.N in AArch64 state.

integer GetNumEventCounters()
    return integer IMPLEMENTATION_DEFINED "Number of event counters";
```

Library pseudocode for shared/debug/pmu/HasElapsed64Cycles

```
// Returns TRUE if 64 cycles have elapsed between the last count, and FALSE otherwise.
boolean HasElapsed64Cycles();
```

Library pseudocode for shared/debug/pmu/PMUCountValue

```
// PMUCountValue()
// ===============
// Implements the PMU threshold function, if implemented.
// Returns the value to increment event counter 'n' by, if the event it is
// configured to count yields the value 'v' on this cycle.

integer PMUCountValue(integer n, integer v)
    if !HavePMUv3TH() then
        return v;

    integer T = UInt(PMEVTYPER_EL0[n].TH);

    case PMEVTYPER_EL0[n].TC of
        when '000' return (if v != T then v else 0);    // Disabled or not-equal
        when '001' return (if v != T then 1 else 0);    // Not-equal, count
        when '010' return (if v == T then v else 0);    // Equals
        when '011' return (if v == T then 1 else 0);    // Equals, count
        when '100' return (if v >= T then v else 0);    // Greater-than-or-equal
        when '101' return (if v >= T then 1 else 0);    // Greater-than-or-equal, count
        when '110' return (if v <  T then v else 0);    // Less-than
        when '111' return (if v <  T then 1 else 0);    // Less-than, count
```

Library pseudocode for shared/debug/pmu/PMUCounterMask

```
constant integer CYCLE_COUNTER_ID = 31;

// PMUCounterMask()
// ================
// Return bitmask of accessible PMU counters.

bits(32) PMUCounterMask()
    integer n;
    if UsingAArch32() then
        n = AArch32.GetNumEventCountersAccessible();
    else
        n = AArch64.GetNumEventCountersAccessible();
    return '1' : ZeroExtend(Ones(n), 31);
```

Library pseudocode for shared/debug/pmu/PMUEvent

```
// PMUEvent()
// ==========
// Generate a PMU event. By default, increment by 1.

PMUEvent(bits(16) event)
    PMUEvent(event, 1);

// PMUEvent()
// ==========
// Accumulate a PMU Event.

PMUEvent(bits(16) event, integer increment)
    integer counters = GetNumEventCounters();
    if counters != 0 then
        for idx = 0 to counters - 1
            PMUEvent(event, increment, idx);

// PMUEvent()
// ==========
// Accumulate a PMU Event for a specific event counter.

PMUEvent(bits(16) event, integer increment, integer idx)
    if !HavePMUv3() then
        return;

    if UsingAArch32() then
        if PMEVTYPER[idx].evtCount == event then
            PMUEventAccumulator[idx] = PMUEventAccumulator[idx] + increment;
    else
        if PMEVTYPER_EL0[idx].evtCount == event then
            PMUEventAccumulator[idx] = PMUEventAccumulator[idx] + increment;
```

Library pseudocode for shared/debug/pmu/integer

```
array integer PMUEventAccumulator[0..30];   // Accumulates PMU events for a cycle
```

Library pseudocode for shared/debug/samplebasedprofiling/CreatePCSample

```
// CreatePCSample()
// ================

CreatePCSample()
    // In a simple sequential execution of the program, CreatePCSample is executed each time the PE
    // executes an instruction that can be sampled. An implementation is not constrained such that
    // reads of EDPCSRlo return the current values of PC, etc.

    pc_sample.valid = ExternalNoninvasiveDebugAllowed() && !Halted();
    pc_sample.pc = ThisInstrAddr(64);
    pc_sample.el = PSTATE.EL;
    pc_sample.rw = if UsingAArch32() then '0' else '1';
    pc_sample.ss = CurrentSecurityState();
    pc_sample.contextidr = if ELUsingAArch32(EL1) then CONTEXTIDR else CONTEXTIDR_EL1<31:0>;
    pc_sample.has_el2 = PSTATE.EL != EL3 && EL2Enabled();

    if pc_sample.has_el2 then
        if ELUsingAArch32(EL2) then
            pc_sample.vmid = ZeroExtend(VTTBR.VMID, 16);
        elsif !Have16bitVMID() || VTCR_EL2.VS == '0' then
            pc_sample.vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
        else
            pc_sample.vmid = VTTBR_EL2.VMID;
        if (HaveVirtHostExt() || HaveV82Debug()) && !ELUsingAArch32(EL2) then
            pc_sample.contextidr_el2 = CONTEXTIDR_EL2<31:0>;
        else
            pc_sample.contextidr_el2 = bits(32) UNKNOWN;
        pc_sample.el0h = PSTATE.EL == EL0 && IsInHost();
    return;
```

Library pseudocode for shared/debug/samplebasedprofiling/EDPCSRlo

```
// EDPCSRlo[] (read)
// ================

bits(32) EDPCSRlo[boolean memory_mapped]

    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "generate error response";
        return bits(32) UNKNOWN;

    // The Software lock is OPTIONAL.
    update = !memory_mapped || EDLSR.SLK == '0';        // Software locked: no side-effects

    bits(32) sample;
    if pc_sample.valid then
        sample = pc_sample.pc<31:0>;
        if update then
            if HaveVirtHostExt() && EDSCR.SC2 == '1' then
                EDPCSRhi.PC = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
                EDPCSRhi.EL = pc_sample.el;
                EDPCSRhi.NS = (if pc_sample.ss == SS_Secure then '0' else '1');
            else
                EDPCSRhi = (if pc_sample.rw == '0' then Zeros(32) else pc_sample.pc<63:32>);
            EDCIDSR = pc_sample.contextidr;
            if (HaveVirtHostExt() || HaveV82Debug()) && EDSCR.SC2 == '1' then
                EDVIDSR = (if pc_sample.has_el2 then pc_sample.contextidr_el2
                           else bits(32) UNKNOWN);
            else
                EDVIDSR.VMID = (if pc_sample.has_el2 && pc_sample.el IN {EL1,EL0}
                                then pc_sample.vmid else Zeros(16));
                EDVIDSR.NS = (if pc_sample.ss == SS_Secure then '0' else '1');
                EDVIDSR.E2 = (if pc_sample.el == EL2 then '1' else '0');
                EDVIDSR.E3 = (if pc_sample.el == EL3 then '1' else '0') AND pc_sample.rw;
                // The conditions for setting HV are not specified if PCSRhi is zero.
                // An example implementation may be "pc_sample.rw".
                EDVIDSR.HV = (if !IsZero(EDPCSRhi) then '1' else bit IMPLEMENTATION_DEFINED "0 or 1");
    else
        sample = Ones(32);
        if update then
            EDPCSRhi = bits(32) UNKNOWN;
            EDCIDSR = bits(32) UNKNOWN;
            EDVIDSR = bits(32) UNKNOWN;

    return sample;
```

Library pseudocode for shared/debug/samplebasedprofiling/PCSample

```
type PCSample is (
    boolean valid,
    bits(64) pc,
    bits(2) el,
    bit rw,
    SecurityState ss,
    boolean has_el2,
    bits(32) contextidr,
    bits(32) contextidr_el2,
    boolean el0h,
    bits(16) vmid
)

PCSample pc_sample;
```

Library pseudocode for shared/debug/samplebasedprofiling/PMPCSR

```
// PMPCSR[] (read)
// ===============

bits(32) PMPCSR[boolean memory_mapped]
    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "generate error response";
        return bits(32) UNKNOWN;

    // The Software lock is OPTIONAL.
    update = !memory_mapped || PMLSR.SLK == '0';        // Software locked: no side-effects

    bits(32) sample;
    if pc_sample.valid then
        sample = pc_sample.pc<31:0>;
        if update then
            PMPCSR<55:32> = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
            PMPCSR.EL = pc_sample.el;
            if HaveRME() then
                case pc_sample.ss of
                    when SS_Secure
                        PMPCSR.NSE = '0'; PMPCSR.NS = '0';
                    when SS_NonSecure
                        PMPCSR.NSE = '0'; PMPCSR.NS = '1';
                    when SS_Root
                        PMPCSR.NSE = '1'; PMPCSR.NS = '0';
                    when SS_Realm
                        PMPCSR.NSE = '1'; PMPCSR.NS = '1';
            else
                PMPCSR.NS = (if pc_sample.ss == SS_Secure then '0' else '1');

            PMCID1SR = pc_sample.contextidr;
            PMCID2SR = if pc_sample.has_el2 then pc_sample.contextidr_el2 else bits(32) UNKNOWN;

            PMVIDSR.VMID = (if pc_sample.has_el2 && pc_sample.el IN {EL1,EL0} && !pc_sample.el0h
                            then pc_sample.vmid else bits(16) UNKNOWN);
    else
        sample = Ones(32);
        if update then
            PMPCSR<55:32>  = bits(24) UNKNOWN;
            PMPCSR.EL = bits(2) UNKNOWN;
            PMPCSR.NS = bit UNKNOWN;

            PMCID1SR = bits(32) UNKNOWN;
            PMCID2SR = bits(32) UNKNOWN;

            PMVIDSR.VMID = bits(16) UNKNOWN;

    return sample;
```

Library pseudocode for shared/debug/softwarestep/CheckSoftwareStep

```
// CheckSoftwareStep()
// ===================
// Take a Software Step exception if in the active-pending state

CheckSoftwareStep()

    // Other self-hosted debug functions will call AArch32.GenerateDebugExceptions() if called from
    // AArch32 state. However, because Software Step is only active when the debug target Exception
    // level is using AArch64, CheckSoftwareStep only calls AArch64.GenerateDebugExceptions().
    step_enabled = !ELUsingAArch32(DebugTarget()) && AArch64.GenerateDebugExceptions() && MDSCR_EL1.SS ==
    if step_enabled && PSTATE.SS == '0' then
        AArch64.SoftwareStepException();
```

Library pseudocode for shared/debug/softwarestep/DebugExceptionReturnSS

```
// DebugExceptionReturnSS()
// =======================
// Returns value to write to PSTATE.SS on an exception return or Debug state exit.

bit DebugExceptionReturnSS(bits(N) spsr)
    if UsingAArch32() then
        assert N == 32;
    else
        assert N == 64;

    assert Halted() || Restarting() ||  PSTATE.EL != EL0;

    boolean enabled_at_source;
    if Restarting() then
        enabled_at_source = FALSE;
    elsif UsingAArch32() then
        enabled_at_source = AArch32.GenerateDebugExceptions();
    else
        enabled_at_source = AArch64.GenerateDebugExceptions();

    boolean valid;
    bits(2) dest_el;
    if IllegalExceptionReturn(spsr) then
        dest_el = PSTATE.EL;
    else
        (valid, dest_el) = ELFromSPSR(spsr);  assert valid;

    dest_ss = SecurityStateAtEL(dest_el);
    bit mask;
    boolean enabled_at_dest;
    dest_using_32 = (if dest_el == EL0 then spsr<4> == '1' else ELUsingAArch32(dest_el));
    if dest_using_32 then
        enabled_at_dest = AArch32.GenerateDebugExceptionsFrom(dest_el, dest_ss);
    else
        mask = spsr<9>;
        enabled_at_dest = AArch64.GenerateDebugExceptionsFrom(dest_el, dest_ss, mask);

    ELd = DebugTargetFrom(dest_ss);
    bit SS_bit;
    if !ELUsingAArch32(ELd) && MDSCR_EL1.SS == '1' && !enabled_at_source && enabled_at_dest then
        SS_bit = spsr<21>;
    else
        SS_bit = '0';

    return SS_bit;
```

Library pseudocode for shared/debug/softwarestep/SSAdvance

```
// SSAdvance()
// ===========
// Advance the Software Step state machine.

SSAdvance()

    // A simpler implementation of this function just clears PSTATE.SS to zero regardless of the
    // current Software Step state machine. However, this check is made to illustrate that the
    // processor only needs to consider advancing the state machine from the active-not-pending
    // state.
    target = DebugTarget();
    step_enabled = !ELUsingAArch32(target) && MDSCR_EL1.SS == '1';
    active_not_pending = step_enabled && PSTATE.SS == '1';

    if active_not_pending then PSTATE.SS = '0';

    return;
```

Library pseudocode for shared/debug/softwarestep/SoftwareStep_DidNotStep

```
// Returns TRUE if the previously executed instruction was executed in the
// inactive state, that is, if it was not itself stepped.
// Might return TRUE or FALSE if the previously executed instruction was an ISB
// or ERET executed in the active-not-pending state, or if another exception
// was taken before the Software Step exception.  Returns FALSE otherwise,
// indicating that the previously executed instruction was executed in the
// active-not-pending state, that is, the instruction was stepped.
boolean SoftwareStep_DidNotStep();
```

Library pseudocode for shared/debug/softwarestep/SoftwareStep_SteppedEX

```
// Returns a value that describes the previously executed instruction. The
// result is valid only if SoftwareStep_DidNotStep() returns FALSE.
// Might return TRUE or FALSE if the instruction was an AArch32 LDREX or LDAEX
// that failed its condition code test.  Otherwise returns TRUE if the
// instruction was a Load-Exclusive class instruction, and FALSE if the
// instruction was not a Load-Exclusive class instruction.
boolean SoftwareStep_SteppedEX();
```

Library pseudocode for shared/exceptions/exceptions/ConditionSyndrome

```
// ConditionSyndrome()
// ===================
// Return CV and COND fields of instruction syndrome

bits(5) ConditionSyndrome()

    bits(5) syndrome;

    if UsingAArch32() then
        cond = AArch32.CurrentCond();
        if PSTATE.T == '0' then             // A32
            syndrome<4> = '1';
            // A conditional A32 instruction that is known to pass its condition code check
            // can be presented either with COND set to 0xE, the value for unconditional, or
            // the COND value held in the instruction.
            if ConditionHolds(cond) && ConstrainUnpredictableBool(Unpredictable_ESRCONDPASS) then
                syndrome<3:0> = '1110';
            else
                syndrome<3:0> = cond;
        else                                // T32
            // When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
            //  * CV set to 0 and COND is set to an UNKNOWN value
            //  * CV set to 1 and COND is set to the condition code for the condition that
            //    applied to the instruction.
            if boolean IMPLEMENTATION_DEFINED "Condition valid for trapped T32" then
                syndrome<4> = '1';
                syndrome<3:0> = cond;
            else
                syndrome<4> = '0';
                syndrome<3:0> = bits(4) UNKNOWN;
    else
        syndrome<4> = '1';
        syndrome<3:0> = '1110';

    return syndrome;
```

Library pseudocode for shared/exceptions/exceptions/Exception

```
enumeration Exception {Exception_Uncategorized,       // Uncategorized or unknown reason
                       Exception_WFxTrap,             // Trapped WFI or WFE instruction
                       Exception_CP15RTTrap,          // Trapped AArch32 MCR or MRC access, coproc=0b111
                       Exception_CP15RRTTrap,         // Trapped AArch32 MCRR or MRRC access, coproc=0b11
                       Exception_CP14RTTrap,          // Trapped AArch32 MCR or MRC access, coproc=0b1110
                       Exception_CP14DTTrap,          // Trapped AArch32 LDC or STC access, coproc=0b1110
                       Exception_CP14RRTTrap,         // Trapped AArch32 MRRC access, coproc=0b1110
                       Exception_AdvSIMDFPAccessTrap, // HCPTR-trapped access to SIMD or FP
                       Exception_FPIDTrap,            // Trapped access to SIMD or FP ID register
                       Exception_LDST64BTrap,         // Trapped access to ST64BV, ST64BV0, ST64B and LD6
                       // Trapped BXJ instruction not supported in Armv8
                       Exception_PACTrap,             // Trapped invalid PAC use
                       Exception_IllegalState,        // Illegal Execution state
                       Exception_SupervisorCall,      // Supervisor Call
                       Exception_HypervisorCall,      // Hypervisor Call
                       Exception_MonitorCall,         // Monitor Call or Trapped SMC instruction
                       Exception_SystemRegisterTrap,  // Trapped MRS or MSR System register access
                       Exception_ERetTrap,            // Trapped invalid ERET use
                       Exception_InstructionAbort,    // Instruction Abort or Prefetch Abort
                       Exception_PCAlignment,         // PC alignment fault
                       Exception_DataAbort,           // Data Abort
                       Exception_NV2DataAbort,        // Data abort at EL1 reported as being from EL2
                       Exception_PACFail,             // PAC Authentication failure
                       Exception_SPAlignment,         // SP alignment fault
                       Exception_FPTrappedException,  // IEEE trapped FP exception
                       Exception_SError,              // SError interrupt
                       Exception_Breakpoint,          // (Hardware) Breakpoint
                       Exception_SoftwareStep,        // Software Step
                       Exception_Watchpoint,          // Watchpoint
                       Exception_NV2Watchpoint,       // Watchpoint at EL1 reported as being from EL2
                       Exception_SoftwareBreakpoint,  // Software Breakpoint Instruction
                       Exception_VectorCatch,         // AArch32 Vector Catch
                       Exception_IRQ,                 // IRQ interrupt
                       Exception_SVEAccessTrap,       // HCPTR trapped access to SVE
                       Exception_SMEAccessTrap,       // HCPTR trapped access to SME
                       Exception_TSTARTAccessTrap,    // Trapped TSTART access
                       Exception_GPC,                 // Granule protection check
                       Exception_BranchTarget,        // Branch Target Identification
                       Exception_MemCpyMemSet,        // Exception from a CPY* or SET* instruction
                       Exception_FIQ};                // FIQ interrupt
```

Library pseudocode for shared/exceptions/exceptions/ExceptionRecord

```
type ExceptionRecord is (
    Exception   exceptype,           // Exception class
    bits(25)    syndrome,            // Syndrome record
    bits(5)     syndrome2,           // ST64BV(0) return value register specifier
    FullAddress paddress,            // Physical fault address
    bits(64)    vaddress,            // Virtual fault address
    boolean     ipavalid,            // Validity of Intermediate Physical fault address
    bit         NS,                  // Intermediate Physical fault address space
    bits(52)    ipaddress,           // Intermediate Physical fault address
    boolean     trappedsyscallinst)  // Trapped SVC or SMC instruction
```

Library pseudocode for shared/exceptions/exceptions/ExceptionSyndrome

```
// ExceptionSyndrome()
// ===================
// Return a blank exception syndrome record for an exception of the given type.

ExceptionRecord ExceptionSyndrome(Exception exceptype)

    ExceptionRecord r;

    r.exceptype = exceptype;

    // Initialize all other fields
    r.syndrome  = Zeros(25);
    r.syndrome2 = Zeros(5);
    r.vaddress  = Zeros(64);
    r.ipavalid  = FALSE;
    r.NS        = '0';
    r.ipaddress = Zeros(52);
    r.paddress.paspace = PASpace UNKNOWN;
    r.paddress.address = bits(52) UNKNOWN;
    r.trappedsyscallinst = FALSE;
    return r;
```

Library pseudocode for shared/functions/aborts/EncodeLDFSC

```
// EncodeLDFSC()
// =============
// Function that gives the Long-descriptor FSC code for types of Fault

bits(6) EncodeLDFSC(Fault statuscode, integer level)
    bits(6) result;

    if level == -1 then
        assert Have52BitIPAAndPASpaceExt();
        case statuscode of
            when Fault_AddressSize          result = '101001';
            when Fault_Translation          result = '101011';
            when Fault_SyncExternalOnWalk   result = '010011';
            when Fault_SyncParityOnWalk     result = '011011'; assert !HaveRASExt();
            when Fault_GPCFOnWalk           result = '100011';
            otherwise                       Unreachable();

        return result;
    case statuscode of
        when Fault_AddressSize         result = '0000':level<1:0>; assert level IN {0,1,2,3};
        when Fault_AccessFlag          result = '0010':level<1:0>; assert level IN {0,1,2,3};
        when Fault_Permission          result = '0011':level<1:0>; assert level IN {0,1,2,3};
        when Fault_Translation         result = '0001':level<1:0>; assert level IN {0,1,2,3};
        when Fault_SyncExternal        result = '010000';
        when Fault_SyncExternalOnWalk  result = '0101':level<1:0>; assert level IN {0,1,2,3};
        when Fault_SyncParity          result = '011000';
        when Fault_SyncParityOnWalk    result = '0111':level<1:0>; assert level IN {0,1,2,3};
        when Fault_AsyncParity         result = '011001';
        when Fault_AsyncExternal       result = '010001';
        when Fault_Alignment           result = '100001';
        when Fault_Debug               result = '100010';
        when Fault_GPCFOnWalk          result = '1001':level<1:0>; assert level IN {0,1,2,3};
        when Fault_GPCFOnOutput        result = '101000';
        when Fault_TLBConflict         result = '110000';
        when Fault_HWUpdateAccessFlag  result = '110001';
        when Fault_Lockdown            result = '110100';  // IMPLEMENTATION DEFINED
        when Fault_Exclusive           result = '110101';  // IMPLEMENTATION DEFINED
        otherwise                      Unreachable();

    return result;
```

Library pseudocode for shared/functions/aborts/IPAValid

```
// IPAValid()
// ==========
// Return TRUE if the IPA is reported for the abort

boolean IPAValid(FaultRecord fault)
    assert fault.statuscode != Fault_None;

    if fault.gpcf.gpf != GPCF_None then
        return fault.secondstage;
    elsif fault.s2fs1walk then
        return fault.statuscode IN {
            Fault_AccessFlag,
            Fault_Permission,
            Fault_Translation,
            Fault_AddressSize
        };
    elsif fault.secondstage then
        return fault.statuscode IN {
            Fault_AccessFlag,
            Fault_Translation,
            Fault_AddressSize
        };
    else
        return FALSE;
```

Library pseudocode for shared/functions/aborts/IsAsyncAbort

```
// IsAsyncAbort()
// ==============
// Returns TRUE if the abort currently being processed is an asynchronous abort, and FALSE
// otherwise.

boolean IsAsyncAbort(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsAsyncAbort()
// ==============

boolean IsAsyncAbort(FaultRecord fault)
    return IsAsyncAbort(fault.statuscode);
```

Library pseudocode for shared/functions/aborts/IsDebugException

```
// IsDebugException()
// ==================

boolean IsDebugException(FaultRecord fault)
    assert fault.statuscode != Fault_None;
    return fault.statuscode == Fault_Debug;
```

Library pseudocode for shared/functions/aborts/IsExternalAbort

```
// IsExternalAbort()
// =================
// Returns TRUE if the abort currently being processed is an External abort and FALSE otherwise.

boolean IsExternalAbort(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {
        Fault_SyncExternal,
        Fault_SyncParity,
        Fault_SyncExternalOnWalk,
        Fault_SyncParityOnWalk,
        Fault_AsyncExternal,
        Fault_AsyncParity
    });

// IsExternalAbort()
// =================

boolean IsExternalAbort(FaultRecord fault)
    return IsExternalAbort(fault.statuscode) || fault.gpcf.gpf == GPCF_EABT;
```

Library pseudocode for shared/functions/aborts/IsExternalSyncAbort

```
// IsExternalSyncAbort()
// =====================
// Returns TRUE if the abort currently being processed is an external
// synchronous abort and FALSE otherwise.

boolean IsExternalSyncAbort(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {
        Fault_SyncExternal,
        Fault_SyncParity,
        Fault_SyncExternalOnWalk,
        Fault_SyncParityOnWalk
    });

// IsExternalSyncAbort()
// =====================

boolean IsExternalSyncAbort(FaultRecord fault)
    return IsExternalSyncAbort(fault.statuscode) || fault.gpcf.gpf == GPCF_EABT;
```

Library pseudocode for shared/functions/aborts/IsFault

```
// IsFault()
// =========
// Return TRUE if a fault is associated with an address descriptor

boolean IsFault(AddressDescriptor addrdesc)
    return addrdesc.fault.statuscode != Fault_None;

// IsFault()
// =========
// Return TRUE if a fault is associated with a memory access.

boolean IsFault(Fault fault)
    return fault != Fault_None;

// IsFault()
// =========
// Return TRUE if a fault is associated with status returned by memory.

boolean IsFault(PhysMemRetStatus retstatus)
    return retstatus.statuscode != Fault_None;
```

Library pseudocode for shared/functions/aborts/IsSErrorInterrupt

```
// IsSErrorInterrupt()
// ===================
// Returns TRUE if the abort currently being processed is an SError interrupt, and FALSE
// otherwise.

boolean IsSErrorInterrupt(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsSErrorInterrupt()
// ===================

boolean IsSErrorInterrupt(FaultRecord fault)
    return IsSErrorInterrupt(fault.statuscode);
```

Library pseudocode for shared/functions/aborts/IsSecondStage

```
// IsSecondStage()
// ===============

boolean IsSecondStage(FaultRecord fault)
    assert fault.statuscode != Fault_None;

    return fault.secondstage;
```

Library pseudocode for shared/functions/aborts/LSInstructionSyndrome

```
// Returns the extended syndrome information for a second stage fault.
//  <10>  - Syndrome valid bit. The syndrome is valid only for certain types of access instruction.
//  <9:8> - Access size.
//  <7>   - Sign extended (for loads).
//  <6:2> - Transfer register.
//  <1>   - Transfer register is 64-bit.
//  <0>   - Instruction has acquire/release semantics.
bits(11) LSInstructionSyndrome();
```

Library pseudocode for shared/functions/aborts/ReportAsGPCException

```
// ReportAsGPCException()
// ======================
// Determine whether the given GPCF is reported as a Granule Protection Check Exception
// rather than a Data or Instruction Abort

boolean ReportAsGPCException(FaultRecord fault)
    assert HaveRME();
    assert fault.statuscode IN {Fault_GPCFOnWalk, Fault_GPCFOnOutput};
    assert fault.gpcf.gpf != GPCF_None;

    case fault.gpcf.gpf of
        when GPCF_Walk        return TRUE;
        when GPCF_AddressSize return TRUE;
        when GPCF_EABT        return TRUE;
        when GPCF_Fail        return SCR_EL3.GPF == '1' && PSTATE.EL != EL3;
```

Library pseudocode for shared/functions/cache/CACHE_OP

```
// CACHE_OP()
// ==========
// Performs Cache maintenance operations as per CacheRecord.

CACHE_OP(CacheRecord cache)
    IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/functions/cache/CPASAtPAS

```
// CPASAtPAS()
// ===========
// Get cache PA space for given PA space.

CachePASpace CPASAtPAS(PASpace pas)
    case pas of
        when PAS_NonSecure
            return CPAS_NonSecure;
        when PAS_Secure
            return CPAS_Secure;
        when PAS_Root
            return CPAS_Root;
        when PAS_Realm
            return CPAS_Realm;
```

Library pseudocode for shared/functions/cache/CPASAtSecurityState

```
// CPASAtSecurityState()
// =====================
// Get cache PA space for given security state.

CachePASpace CPASAtSecurityState(SecurityState ss)
    case ss of
        when SS_NonSecure
            return CPAS_NonSecure;
        when SS_Secure
            return CPAS_SecureNonSecure;
        when SS_Root
            return CPAS_Any;
        when SS_Realm
            return CPAS_RealmNonSecure;
```

Library pseudocode for shared/functions/cache/CacheOp

```
enumeration CacheOp {
    CacheOp_Clean,
    CacheOp_Invalidate,
    CacheOp_CleanInvalidate
};
```

Library pseudocode for shared/functions/cache/CacheOpScope

```
enumeration CacheOpScope {
    CacheOpScope_SetWay,
    CacheOpScope_PoU,
    CacheOpScope_PoC,
    CacheOpScope_PoP,
    CacheOpScope_PoDP,
    CacheOpScope_ALLU,
    CacheOpScope_ALLUIS
};
```

Library pseudocode for shared/functions/cache/CachePASpace

```
enumeration CachePASpace {
    CPAS_NonSecure,
    CPAS_Any,               // match entries from any PA Space - possible from Root only
    CPAS_RealmNonSecure,    // match entries from Realm or Non-Secure PAS
    CPAS_Realm,
    CPAS_Root,
    CPAS_SecureNonSecure,   // match entries from Secure or Non-Secure PAS
    CPAS_Secure
};
```

Library pseudocode for shared/functions/cache/CacheRecord

```
type CacheRecord is (
    AccType          acctype,           // Access type
    CacheOp          cacheop,           // Cache operation
    CacheOpScope     opscope,           // Cache operation type
    CacheType        cachetype,         // Cache type
    bits(64)         regval,
    FullAddress      paddress,
    bits(64)         vaddress,          // For VA operations
    integer          set,               // For SW operations
    integer          way,               // For SW operations
    integer          level,             // For SW operations
    Shareability     shareability,
    boolean          translated,
    boolean          is_vmid_valid,     // is vmid valid for current context
    bits(16)         vmid,
    boolean          is_asid_valid,     // is asid valid for current context
    bits(16)         asid,
    SecurityState    security,
    // For cache operations to full cache or by set/way
    // For operations by address, PA space in paddress
    CachePASpace     cpas
)
```

Library pseudocode for shared/functions/cache/CacheType

```
enumeration CacheType {
    CacheType_Data,
    CacheType_Tag,
    CacheType_Data_Tag,
    CacheType_Instruction
};
```

Library pseudocode for shared/functions/cache/DCInstNeedsTranslation

```
// DCInstNeedsTranslation()
// =======================
// Check whether Data Cache operation needs translation.

boolean DCInstNeedsTranslation(CacheOpScope opscope)
    if CLIDR_EL1.LoC == '000' then
        return !boolean IMPLEMENTATION_DEFINED "No fault generated for DC operations if PoC is before any

    if CLIDR_EL1.LoUU == '000' && opscope == CacheOpScope_PoU then
        return !boolean IMPLEMENTATION_DEFINED "No fault generated for DC operations if PoU is before any

    return TRUE;
```

Library pseudocode for shared/functions/cache/DecodeSW

```
// DecodeSW()
// ==========
// Decode input value into set, way and level for SW instructions.

(integer, integer, integer) DecodeSW(bits(64) regval, CacheType cachetype)
    level = UInt(regval[3:1]);
    (set, way, linesize) = GetCacheInfo(level, cachetype);
    return (set, way, level);
```

Library pseudocode for shared/functions/cache/GetCacheInfo

```
// Returns numsets, assosciativity & linesize.
(integer, integer, integer) GetCacheInfo(integer level, CacheType cachetype);
```

Library pseudocode for shared/functions/cache/ICInstNeedsTranslation

```
// ICInstNeedsTranslation()
// =======================
// Check whether Instruction Cache operation needs translation.

boolean ICInstNeedsTranslation(CacheOpScope opscope)
    return boolean IMPLEMENTATION_DEFINED "Instruction Cache needs translation";
```

Library pseudocode for shared/functions/common/ASR

```
// ASR()
// =====

bits(N) ASR(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = ASR_C(x, shift);
    return result;
```

Library pseudocode for shared/functions/common/ASR_C

```
// ASR_C()
// =======

(bits(N), bit) ASR_C(bits(N) x, integer shift)
    assert shift > 0 && shift < 256;
    extended_x = SignExtend(x, shift+N);
    result = extended_x<shift+N-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);
```

Library pseudocode for shared/functions/common/Abs

```
// Abs()
// =====

integer Abs(integer x)
    return if x >= 0 then x else -x;

// Abs()
// =====

real Abs(real x)
    return if x >= 0.0 then x else -x;
```

Library pseudocode for shared/functions/common/Align

```
// Align()
// =======

integer Align(integer x, integer y)
    return y * (x DIV y);

// Align()
// =======

bits(N) Align(bits(N) x, integer y)
    return Align(UInt(x), y)<N-1:0>;
```

Library pseudocode for shared/functions/common/BitCount

```
// BitCount()
// ==========

integer BitCount(bits(N) x)
    integer result = 0;
    for i = 0 to N-1
        if x<i> == '1' then
            result = result + 1;
    return result;
```

Library pseudocode for shared/functions/common/CountLeadingSignBits

```
// CountLeadingSignBits()
// ======================

integer CountLeadingSignBits(bits(N) x)
    return CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>);
```

Library pseudocode for shared/functions/common/CountLeadingZeroBits

```
// CountLeadingZeroBits()
// ======================

integer CountLeadingZeroBits(bits(N) x)
    return N - (HighestSetBit(x) + 1);
```

Library pseudocode for shared/functions/common/Elem

```
// Elem[] - non-assignment form
// ============================

bits(size) Elem[bits(N) vector, integer e, integer size]
    assert e >= 0 && (e+1)*size <= N;
    return vector<e*size+size-1 : e*size>;

// Elem[] - assignment form
// ========================

Elem[bits(N) &vector, integer e, integer size] = bits(size) value
    assert e >= 0 && (e+1)*size <= N;
    vector<(e+1)*size-1:e*size> = value;
    return;
```

Library pseudocode for shared/functions/common/Extend

```
// Extend()
// ========

bits(N) Extend(bits(M) x, integer N, boolean unsigned)
    return if unsigned then ZeroExtend(x, N) else SignExtend(x, N);
```

Library pseudocode for shared/functions/common/HighestSetBit

```
// HighestSetBit()
// ===============

integer HighestSetBit(bits(N) x)
    for i = N-1 downto 0
        if x<i> == '1' then return i;
    return -1;
```

Library pseudocode for shared/functions/common/Int

```
// Int()
// =====

integer Int(bits(N) x, boolean unsigned)
    result = if unsigned then UInt(x) else SInt(x);
    return result;
```

Library pseudocode for shared/functions/common/IsOnes

```
// IsOnes()
// ========

boolean IsOnes(bits(N) x)
    return x == Ones(N);
```

Library pseudocode for shared/functions/common/IsZero

```
// IsZero()
// ========

boolean IsZero(bits(N) x)
    return x == Zeros(N);
```

Library pseudocode for shared/functions/common/IsZeroBit

```
// IsZeroBit()
// ===========

bit IsZeroBit(bits(N) x)
    return if IsZero(x) then '1' else '0';
```

Library pseudocode for shared/functions/common/LSL

```
// LSL()
// =====

bits(N) LSL(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSL_C(x, shift);
    return result;
```

Library pseudocode for shared/functions/common/LSL_C

```
// LSL_C()
// =======

(bits(N), bit) LSL_C(bits(N) x, integer shift)
    assert shift > 0 && shift < 256;
    extended_x = x : Zeros(shift);
    result = extended_x<N-1:0>;
    carry_out = extended_x<N>;
    return (result, carry_out);
```

Library pseudocode for shared/functions/common/LSR

```
// LSR()
// =====

bits(N) LSR(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSR_C(x, shift);
    return result;
```

Library pseudocode for shared/functions/common/LSR_C

```
// LSR_C()
// =======

(bits(N), bit) LSR_C(bits(N) x, integer shift)
    assert shift > 0 && shift < 256;
    extended_x = ZeroExtend(x, shift+N);
    result = extended_x<shift+N-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);
```

Library pseudocode for shared/functions/common/LowestSetBit

```
// LowestSetBit()
// ==============

integer LowestSetBit(bits(N) x)
    for i = 0 to N-1
        if x<i> == '1' then return i;
    return N;
```

Library pseudocode for shared/functions/common/Max

```
// Max()
// =====

integer Max(integer a, integer b)
    return if a >= b then a else b;

// Max()
// =====

real Max(real a, real b)
    return if a >= b then a else b;
```

Library pseudocode for shared/functions/common/Min

```
// Min()
// =====

integer Min(integer a, integer b)
    return if a <= b then a else b;

// Min()
// =====

real Min(real a, real b)
    return if a <= b then a else b;
```

Library pseudocode for shared/functions/common/Ones

```
// Ones()
// ======

bits(N) Ones(integer N)
    return Replicate('1',N);

// Ones()
// ======

bits(N) Ones()
    return Ones(N);
```

Library pseudocode for shared/functions/common/ROR

```
// ROR()
// =====

bits(N) ROR(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = ROR_C(x, shift);
    return result;
```

Library pseudocode for shared/functions/common/ROR_C

```
// ROR_C()
// =======

(bits(N), bit) ROR_C(bits(N) x, integer shift)
    assert shift != 0 && shift < 256;
    m = shift MOD N;
    result = LSR(x,m) OR LSL(x,N-m);
    carry_out = result<N-1>;
    return (result, carry_out);
```

Library pseudocode for shared/functions/common/Replicate

```
bits(M*N) Replicate(bits(M) x, integer N);
```

Library pseudocode for shared/functions/common/RoundDown

```
integer RoundDown(real x);
```

Library pseudocode for shared/functions/common/RoundTowardsZero

```
// RoundTowardsZero()
// ==================

integer RoundTowardsZero(real x)
    return if x == 0.0 then 0 else if x >= 0.0 then RoundDown(x) else RoundUp(x);
```

Library pseudocode for shared/functions/common/RoundUp

```
integer RoundUp(real x);
```

Library pseudocode for shared/functions/common/SInt

```
// SInt()
// ======

integer SInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    if x<N-1> == '1' then result = result - 2^N;
    return result;
```

Library pseudocode for shared/functions/common/SignExtend

```
// SignExtend()
// ============

bits(N) SignExtend(bits(M) x, integer N)
    assert N >= M;
    return Replicate(x<M-1>, N-M) : x;
```

Library pseudocode for shared/functions/common/Split64to32

```
// Split64to32()
// =============

(bits(32), bits(32)) Split64to32(bits(64) value)
    return (value<63:32>, value<31:0>);
```

Library pseudocode for shared/functions/common/UInt

```
// UInt()
// ======

integer UInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    return result;
```

Library pseudocode for shared/functions/common/ZeroExtend

```
// ZeroExtend()
// ============

bits(N) ZeroExtend(bits(M) x, integer N)
    assert N >= M;
    return Zeros(N-M) : x;
```

Library pseudocode for shared/functions/common/Zeros

```
// Zeros()
// =======

bits(N) Zeros(integer N)
    return Replicate('0',N);

// Zeros()
// =======

bits(N) Zeros()
    return Zeros(N);
```

Library pseudocode for shared/functions/counters/AArch32.CheckTimerConditions

```
// AArch32.CheckTimerConditions()
// ==============================
// Checking timer conditions for all A32 timer registers

AArch32.CheckTimerConditions()
    boolean status;
    bits(64) offset;
    offset = Zeros(64);
    assert !HaveAArch64();

    if HaveEL(EL3) then
        if CNTP_CTL_S.ENABLE == '1' then
            status = IsTimerConditionMet(offset, CNTP_CVAL_S,
                                         CNTP_CTL_S.IMASK, InterruptID_CNTPS);
            CNTP_CTL_S.ISTATUS = if status then '1' else '0';

        if CNTP_CTL_NS.ENABLE == '1' then
            status = IsTimerConditionMet(offset, CNTP_CVAL_NS,
                                         CNTP_CTL_NS.IMASK, InterruptID_CNTP);
            CNTP_CTL_NS.ISTATUS = if status then '1' else '0';
    else
        if CNTP_CTL.ENABLE == '1' then
            status = IsTimerConditionMet(offset, CNTP_CVAL,
                                         CNTP_CTL.IMASK, InterruptID_CNTP);
            CNTP_CTL.ISTATUS = if status then '1' else '0';

    if HaveEL(EL2) && CNTHP_CTL.ENABLE == '1' then
        status = IsTimerConditionMet(offset, CNTHP_CVAL,
                                     CNTHP_CTL.IMASK, InterruptID_CNTHP);
        CNTHP_CTL.ISTATUS = if status then '1' else '0';

    if CNTV_CTL_EL0.ENABLE == '1' then
        status = IsTimerConditionMet(CNTVOFF_EL2, CNTV_CVAL_EL0,
                                     CNTV_CTL_EL0.IMASK, InterruptID_CNTV);
        CNTV_CTL_EL0.ISTATUS = if status then '1' else '0';

    return;
```

Library pseudocode for shared/functions/counters/AArch64.CheckTimerConditions

```
// AArch64.CheckTimerConditions()
// ==============================
// Checking timer conditions for all A64 timer registers

AArch64.CheckTimerConditions()
    boolean status;
    bits(64) offset;
    bit imask;
    SecurityState ss = CurrentSecurityState();
    boolean ecv = FALSE;
    if HaveECVExt() then
        ecv = CNTHCTL_EL2.ECV == '1' && SCR_EL3.ECVEn == '1' && EL2Enabled();
    if ecv then
        offset = CNTPOFF_EL2;
    else
        offset = Zeros(64);
    if CNTP_CTL_EL0.ENABLE == '1' then
        imask = CNTP_CTL_EL0.IMASK;
        if HaveRME() && ss IN {SS_Root, SS_Realm} && CNTHCTL_EL2.CNTPMASK == '1' then
            imask = '1';
        status = IsTimerConditionMet(offset, CNTP_CVAL_EL0,
                                     imask, InterruptID_CNTP);
        CNTP_CTL_EL0.ISTATUS = if status then '1' else '0';
    if ((HaveEL(EL3) || (HaveEL(EL2) && !HaveSecureEL2Ext())) &&
       CNTHP_CTL_EL2.ENABLE == '1') then
        status = IsTimerConditionMet(Zeros(64), CNTHP_CVAL_EL2,
                                     CNTHP_CTL_EL2.IMASK, InterruptID_CNTHP);
        CNTHP_CTL_EL2.ISTATUS = if status then '1' else '0';
    if HaveEL(EL2) && HaveSecureEL2Ext() && CNTHPS_CTL_EL2.ENABLE == '1' then
        status = IsTimerConditionMet(Zeros(64), CNTHPS_CVAL_EL2,
                                     CNTHPS_CTL_EL2.IMASK, InterruptID_CNTHPS);
        CNTHPS_CTL_EL2.ISTATUS = if status then '1' else '0';

    if CNTPS_CTL_EL1.ENABLE == '1' then
        status = IsTimerConditionMet(offset, CNTPS_CVAL_EL1,
                                     CNTPS_CTL_EL1.IMASK, InterruptID_CNTPS);
        CNTPS_CTL_EL1.ISTATUS = if status then '1' else '0';

    if CNTV_CTL_EL0.ENABLE == '1' then
        imask = CNTV_CTL_EL0.IMASK;
        if HaveRME() && ss IN {SS_Root, SS_Realm} && CNTHCTL_EL2.CNTVMASK == '1' then
            imask = '1';
        status = IsTimerConditionMet(CNTVOFF_EL2, CNTV_CVAL_EL0,
                                     imask, InterruptID_CNTV);
        CNTV_CTL_EL0.ISTATUS = if status then '1' else '0';

    if ((HaveVirtHostExt() && (HaveEL(EL3) || !HaveSecureEL2Ext())) &&
        CNTHV_CTL_EL2.ENABLE == '1') then
        status = IsTimerConditionMet(Zeros(64), CNTHV_CVAL_EL2,
                                     CNTHV_CTL_EL2.IMASK, InterruptID_CNTHV);
        CNTHV_CTL_EL2.ISTATUS = if status then '1' else '0';

    if ((HaveSecureEL2Ext() && HaveVirtHostExt()) &&
        CNTHVS_CTL_EL2.ENABLE == '1') then
        status = IsTimerConditionMet(Zeros(64), CNTHVS_CVAL_EL2,
                                     CNTHVS_CTL_EL2.IMASK, InterruptID_CNTHVS);
        CNTHVS_CTL_EL2.ISTATUS = if status then '1' else '0';
    return;
```

Library pseudocode for shared/functions/counters/GenericCounterTick

```
// GenericCounterTick()
// ====================
// Increments PhysicalCount value for every clock tick.

GenericCounterTick()
    bits(64) prev_physical_count;
    if CNTCR.EN == '0' then
        if !HaveAArch64() then
            AArch32.CheckTimerConditions();
        else
            AArch64.CheckTimerConditions();
        return;
    prev_physical_count = PhysicalCountInt();
    if HaveCNTSCExt() && CNTCR.SCEN == '1' then
        PhysicalCount = PhysicalCount + ZeroExtend(CNTSCR, 88);
    else
        PhysicalCount<87:24> = PhysicalCount<87:24> + 1;
    if !HaveAArch64() then
        AArch32.CheckTimerConditions();
    else
        AArch64.CheckTimerConditions();
    TestEventCNTP(prev_physical_count, PhysicalCountInt());
    TestEventCNTV(prev_physical_count, PhysicalCountInt());
    return;
```

Library pseudocode for shared/functions/counters/IsTimerConditionMet

```
// IsTimerConditionMet()
// =====================

boolean IsTimerConditionMet(bits(64) offset, bits(64) compare_value,
                            bits(1) imask, InterruptID intid)
    boolean conditon_met;
    signal level;
    condition_met = (UInt(PhysicalCountInt() - offset) -
                                        UInt(compare_value)) >= 0;
    level = if condition_met && imask == '0' then HIGH else LOW;
    SetInterruptRequestLevel(intid, level);
    return condition_met;
```

Library pseudocode for shared/functions/counters/PhysicalCount

```
bits(88) PhysicalCount;
```

Library pseudocode for shared/functions/counters/SetEventRegister

```
// SetEventRegister()
// ==================
// Sets the Event Register of this PE

SetEventRegister()
    EventRegister = '1';
    return;
```

Library pseudocode for shared/functions/counters/TestEventCNTP

```
// TestEventCNTP()
// ===============
// Generate Event stream from the physical counter

TestEventCNTP(bits(64) prev_physical_count, bits(64) current_physical_count)
    bits(64) offset;
    bits(1) samplebit, previousbit;
    if CNTHCTL_EL2.EVNTEN == '1' then
        n = UInt(CNTHCTL_EL2.EVNTI);
        if HaveECVExt() && CNTHCTL_EL2.EVNTIS == '1' then
            n = n + 8;
        boolean ecv = FALSE;
        if HaveECVExt() then
            ecv = (EL2Enabled() && CNTHCTL_EL2.ECV == '1' &&
                   SCR_EL3.ECVEn == '1');
            offset = if ecv then CNTPOFF_EL2 else Zeros(64);
        samplebit   = (current_physical_count - offset)<n>;
        previousbit = (prev_physical_count - offset)<n>;
        if CNTHCTL_EL2.EVNTDIR == '0' then
            if previousbit == '0' && samplebit == '1' then SetEventRegister();
        else
            if previousbit == '1' && samplebit == '0' then SetEventRegister();
    return;
```

Library pseudocode for shared/functions/counters/TestEventCNTV

```
// TestEventCNTV()
// ===============
// Generate Event stream from the virtual counter

TestEventCNTV(bits(64) prev_physical_count, bits(64) current_physical_count)
    bits(64) offset;
    bits(1) samplebit, previousbit;
    if (!(HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11') &&
        CNTKCTL_EL1.EVNTEN == '1') then
        n = UInt(CNTKCTL_EL1.EVNTI);
        if HaveECVExt() && CNTKCTL_EL1.EVNTIS == '1' then
            n = n + 8;
        if HaveEL(EL2) && (!EL2Enabled() || HCR_EL2.<E2H,TGE> != '11') then
            offset = CNTVOFF_EL2;
        else
            offset = Zeros(64);
        samplebit   = (current_physical_count - offset)<n>;
        previousbit = (prev_physical_count - offset)<n>;
        if CNTKCTL_EL1.EVNTDIR == '0' then
            if previousbit == '0' && samplebit == '1' then SetEventRegister();
        else
            if previousbit == '1' && samplebit == '0' then SetEventRegister();
    return;
```

Library pseudocode for shared/functions/crc/BitReverse

```
// BitReverse()
// ============

bits(N) BitReverse(bits(N) data)
    bits(N) result;
    for i = 0 to N-1
        result<(N-i)-1> = data<i>;
    return result;
```

Library pseudocode for shared/functions/crc/HaveCRCExt

```
// HaveCRCExt()
// ============

boolean HaveCRCExt()
    return HasArchVersion(ARMv8p1) || boolean IMPLEMENTATION_DEFINED "Have CRC extension";
```

Library pseudocode for shared/functions/crc/Poly32Mod2

```
// Poly32Mod2()
// ============

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation

bits(32) Poly32Mod2(bits(N) data_in, bits(32) poly)
    assert N > 32;
    bits(N) data = data_in;
    for i = N-1 downto 32
        if data<i> == '1' then
            data<i-1:0> = data<i-1:0> EOR (poly:Zeros(i-32));
    return data<31:0>;
```

Library pseudocode for shared/functions/crypto/AESInvMixColumns

```
// AESInvMixColumns()
// ==================
// Transformation in the Inverse Cipher that is the inverse of AESMixColumns.

bits(128) AESInvMixColumns(bits (128) op)
    bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op<  0+:8>;
    bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op<  8+:8>;
    bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;
    bits(4*8) in3 = op<120+:8> : op< 88+:8> : op< 56+:8> : op< 24+:8>;

    bits(4*8) out0;
    bits(4*8) out1;
    bits(4*8) out2;
    bits(4*8) out3;

    for c = 0 to 3
        out0<c*8+:8> =  FFmul0E(in0<c*8+:8>) EOR FFmul0B(in1<c*8+:8>) EOR FFmul0D(in2<c*8+:8>) EOR FFmul0
        out1<c*8+:8> =  FFmul09(in0<c*8+:8>) EOR FFmul0E(in1<c*8+:8>) EOR FFmul0B(in2<c*8+:8>) EOR FFmul0
        out2<c*8+:8> =  FFmul0D(in0<c*8+:8>) EOR FFmul09(in1<c*8+:8>) EOR FFmul0E(in2<c*8+:8>) EOR FFmul0
        out3<c*8+:8> =  FFmul0B(in0<c*8+:8>) EOR FFmul0D(in1<c*8+:8>) EOR FFmul09(in2<c*8+:8>) EOR FFmul0

    return (
        out3<3*8+:8> : out2<3*8+:8> : out1<3*8+:8> : out0<3*8+:8> :
        out3<2*8+:8> : out2<2*8+:8> : out1<2*8+:8> : out0<2*8+:8> :
        out3<1*8+:8> : out2<1*8+:8> : out1<1*8+:8> : out0<1*8+:8> :
        out3<0*8+:8> : out2<0*8+:8> : out1<0*8+:8> : out0<0*8+:8>
    );
```

Library pseudocode for shared/functions/crypto/AESInvShiftRows

```
// AESInvShiftRows()
// =================
// Transformation in the Inverse Cipher that is inverse of AESShiftRows.

bits(128) AESInvShiftRows(bits(128) op)
    return (
        op< 31: 24> : op< 55: 48> : op< 79: 72> : op<103: 96> :
        op<127:120> : op< 23: 16> : op< 47: 40> : op< 71: 64> :
        op< 95: 88> : op<119:112> : op< 15:  8> : op< 39: 32> :
        op< 63: 56> : op< 87: 80> : op<111:104> : op<  7:  0>
    );
```

Library pseudocode for shared/functions/crypto/AESInvSubBytes

```
// AESInvSubBytes()
// ================
// Transformation in the Inverse Cipher that is the inverse of AESSubBytes.

bits(128) AESInvSubBytes(bits(128) op)
    // Inverse S-box values
    bits(16*16*8) GF2_inv = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x7d0c2155631469e126d677ba7e042b17<127:0> :
        /*E*/ 0x619953833cbbebc8b0f52aae4d3be0a0<127:0> :
        /*D*/ 0xef9cc9939f7ae52d0d4ab519a97f5160<127:0> :
        /*C*/ 0x5fec8027591012b131c7078833a8dd1f<127:0> :
        /*B*/ 0xf45acd78fec0db9a2079d2c64b3e56fc<127:0> :
        /*A*/ 0x1bbe18aa0e62b76f89c5291d711af147<127:0> :
        /*9*/ 0x6edf751ce837f9e28535ade72274ac96<127:0> :
        /*8*/ 0x73e6b4f0cecff297eadc674f4111913a<127:0> :
        /*7*/ 0x6b8a130103bdafc1020f3fca8f1e2cd0<127:0> :
        /*6*/ 0x0645b3b80558e4f70ad3bc8c00abd890<127:0> :
        /*5*/ 0x849d8da75746155edab9edfd5048706c<127:0> :
        /*4*/ 0x92b6655dcc5ca4d41698688664f6f872<127:0> :
        /*3*/ 0x25d18b6d49a25b76b224d92866a12e08<127:0> :
        /*2*/ 0x4ec3fa420b954cee3d23c2a632947b54<127:0> :
        /*1*/ 0xcbe9dec444438e3487ff2f9b8239e37c<127:0> :
        /*0*/ 0xfbd7f3819ea340bf38a53630d56a0952<127:0>
    );
    bits(128) out;
    for i = 0 to 15
        out<i*8+:8> = GF2_inv<UInt(op<i*8+:8>)*8+:8>;
    return out;
```

Library pseudocode for shared/functions/crypto/AESMixColumns

```
// AESMixColumns()
// ===============
// Transformation in the Cipher that takes all of the columns of the
// State and mixes their data (independently of one another) to
// produce new columns.

bits(128) AESMixColumns(bits (128) op)
    bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op<  0+:8>;
    bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op<  8+:8>;
    bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;
    bits(4*8) in3 = op<120+:8> : op< 88+:8> : op< 56+:8> : op< 24+:8>;

    bits(4*8) out0;
    bits(4*8) out1;
    bits(4*8) out2;
    bits(4*8) out3;

    for c = 0 to 3
        out0<c*8+:8> =  FFmul02(in0<c*8+:8>) EOR FFmul03(in1<c*8+:8>) EOR         in2<c*8+:8>  EOR
        out1<c*8+:8> =          in0<c*8+:8>  EOR FFmul02(in1<c*8+:8>) EOR FFmul03(in2<c*8+:8>) EOR
        out2<c*8+:8> =          in0<c*8+:8>  EOR         in1<c*8+:8>  EOR FFmul02(in2<c*8+:8>) EOR FFmul0
        out3<c*8+:8> =  FFmul03(in0<c*8+:8>) EOR         in1<c*8+:8>  EOR         in2<c*8+:8>  EOR FFmul0

    return (
        out3<3*8+:8> : out2<3*8+:8> : out1<3*8+:8> : out0<3*8+:8> :
        out3<2*8+:8> : out2<2*8+:8> : out1<2*8+:8> : out0<2*8+:8> :
        out3<1*8+:8> : out2<1*8+:8> : out1<1*8+:8> : out0<1*8+:8> :
        out3<0*8+:8> : out2<0*8+:8> : out1<0*8+:8> : out0<0*8+:8>
    );
```

Library pseudocode for shared/functions/crypto/AESShiftRows

```
// AESShiftRows()
// ==============
// Transformation in the Cipher that processes the State by cyclically
// shifting the last three rows of the State by different offsets.

bits(128) AESShiftRows(bits(128) op)
    return (
        op< 95: 88> : op< 55: 48> : op< 15:  8> : op<103: 96> :
        op< 63: 56> : op< 23: 16> : op<111:104> : op< 71: 64> :
        op< 31: 24> : op<119:112> : op< 79: 72> : op< 39: 32> :
        op<127:120> : op< 87: 80> : op< 47: 40> : op<  7:  0>
    );
```

Library pseudocode for shared/functions/crypto/AESSubBytes

```
// AESSubBytes()
// =============
// Transformation in the Cipher that processes the State using a nonlinear
// byte substitution table (S-box) that operates on each of the State bytes
// independently.

bits(128) AESSubBytes(bits(128) op)
    // S-box values
    bits(16*16*8) GF2 = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x16bb54b00f2d99416842e6bf0d89a18c<127:0> :
        /*E*/ 0xdf2855cee9871e9b948ed9691198f8e1<127:0> :
        /*D*/ 0x9e1dc186b95735610ef6034866b53e70<127:0> :
        /*C*/ 0x8a8bbd4b1f74dde8c6b4a61c2e2578ba<127:0> :
        /*B*/ 0x08ae7a65eaf4566ca94ed58d6d37c8e7<127:0> :
        /*A*/ 0x79e4959162acd3c25c2406490a3a32e0<127:0> :
        /*9*/ 0xdb0b5ede14b8ee4688902a22dc4f8160<127:0> :
        /*8*/ 0x73195d643d7ea7c41744975fec130ccd<127:0> :
        /*7*/ 0xd2f3ff1021dab6bcf5389d928f40a351<127:0> :
        /*6*/ 0xa89f3c507f02f94585334d43fbaaefd0<127:0> :
        /*5*/ 0xcf584c4a39becb6a5bb1fc20ed00d153<127:0> :
        /*4*/ 0x842fe329b3d63b52a05a6e1b1a2c8309<127:0> :
        /*3*/ 0x75b227ebe28012079a059618c323c704<127:0> :
        /*2*/ 0x1531d871f1e5a534ccf73f362693fdb7<127:0> :
        /*1*/ 0xc072a49cafa2d4adf04759fa7dc982ca<127:0> :
        /*0*/ 0x76abd7fe2b670130c56f6bf27b777c63<127:0>
    );
    bits(128) out;
    for i = 0 to 15
        out<i*8+:8> = GF2<UInt(op<i*8+:8>)*8+:8>;
    return out;
```

Library pseudocode for shared/functions/crypto/FFmul02

```
// FFmul02()
// =========

bits(8) FFmul02(bits(8) b)
    bits(256*8) FFmul_02 = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0xE5E7E1E3EDEFE9EBF5F7F1F3FDFFF9FB<127:0> :
        /*E*/ 0xC5C7C1C3CDCFC9CBD5D7D1D3DDDFD9DB<127:0> :
        /*D*/ 0xA5A7A1A3ADAFA9ABB5B7B1B3BDBFB9BB<127:0> :
        /*C*/ 0x858781838D8F898B959791939D9F999B<127:0> :
        /*B*/ 0x656761636D6F696B757771737D7F797B<127:0> :
        /*A*/ 0x454741434D4F494B555751535D5F595B<127:0> :
        /*9*/ 0x252721232D2F292B353731333D3F393B<127:0> :
        /*8*/ 0x050701030D0F090B151711131D1F191B<127:0> :
        /*7*/ 0xFEFCFAF8F6F4F2F0EEECEAE8E6E4E2E0<127:0> :
        /*6*/ 0xDEDCDAD8D6D4D2D0CECCCAC8C6C4C2C0<127:0> :
        /*5*/ 0xBEBCBAB8B6B4B2B0AEACAAA8A6A4A2A0<127:0> :
        /*4*/ 0x9E9C9A98969492908E8C8A8886848280<127:0> :
        /*3*/ 0x7E7C7A78767472706E6C6A6866646260<127:0> :
        /*2*/ 0x5E5C5A58565452504E4C4A4846444240<127:0> :
        /*1*/ 0x3E3C3A38363432302E2C2A2826242220<127:0> :
        /*0*/ 0x1E1C1A18161412100E0C0A0806040200<127:0>
    );
    return FFmul_02<UInt(b)*8+:8>;
```

Library pseudocode for shared/functions/crypto/FFmul03

```
// FFmul03()
// =========

bits(8) FFmul03(bits(8) b)
    bits(256*8) FFmul_03 = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x1A191C1F16151013020104070E0D080B<127:0> :
        /*E*/ 0x2A292C2F26252023323134373E3D383B<127:0> :
        /*D*/ 0x7A797C7F76757073626164676E6D686B<127:0> :
        /*C*/ 0x4A494C4F46454043525154575E5D585B<127:0> :
        /*B*/ 0xDAD9DCDFD6D5D0D3C2C1C4C7CECDC8CB<127:0> :
        /*A*/ 0xEAE9ECEFE6E5E0E3F2F1F4F7FEFDF8FB<127:0> :
        /*9*/ 0xBAB9BCBFB6B5B0B3A2A1A4A7AEADA8AB<127:0> :
        /*8*/ 0x8A898C8F86858083929194979E9D989B<127:0> :
        /*7*/ 0x818287848D8E8B88999A9F9C95969390<127:0> :
        /*6*/ 0xB1B2B7B4BDBEBBB8A9AAAFACA5A6A3A0<127:0> :
        /*5*/ 0xE1E2E7E4EDEEEBE8F9FAFFFCF5F6F3F0<127:0> :
        /*4*/ 0xD1D2D7D4DDDEDBD8C9CACFCCC5C6C3C0<127:0> :
        /*3*/ 0x414247444D4E4B48595A5F5C55565350<127:0> :
        /*2*/ 0x717277747D7E7B78696A6F6C65666360<127:0> :
        /*1*/ 0x212227242D2E2B28393A3F3C35363330<127:0> :
        /*0*/ 0x111217141D1E1B18090A0F0C05060300<127:0>
    );
    return FFmul_03<UInt(b)*8+:8>;
```

Library pseudocode for shared/functions/crypto/FFmul09

```
// FFmul09()
// =========

bits(8) FFmul09(bits(8) b)
    bits(256*8) FFmul_09 = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x464F545D626B70790E071C152A233831<127:0> :
        /*E*/ 0xD6DFC4CDF2FBE0E99E978C85BAB3A8A1<127:0> :
        /*D*/ 0x7D746F6659504B42353C272E1118030A<127:0> :
        /*C*/ 0xEDE4FFF6C9C0DBD2A5ACB7BE8188939A<127:0> :
        /*B*/ 0x3039222B141D060F78716A635C554E47<127:0> :
        /*A*/ 0xA0A9B2BB848D969FE8E1FAF3CCC5DED7<127:0> :
        /*9*/ 0x0B0219102F263D34434A5158676E757C<127:0> :
        /*8*/ 0x9B928980BFB6ADA4D3DAC1C8F7FEE5EC<127:0> :
        /*7*/ 0xAAA3B8B18E879C95E2EBF0F9C6CFD4DD<127:0> :
        /*6*/ 0x3A3328211E170C05727B6069565F444D<127:0> :
        /*5*/ 0x9198838AB5BCA7AED9D0CBC2FDF4EFE6<127:0> :
        /*4*/ 0x0108131A252C373E49405B526D647F76<127:0> :
        /*3*/ 0xDCD5CEC7F8F1EAE3949D868FB0B9A2AB<127:0> :
        /*2*/ 0x4C455E5768617A73040D161F2029323B<127:0> :
        /*1*/ 0xE7EEF5FCC3CAD1D8AFA6BDB48B829990<127:0> :
        /*0*/ 0x777E656C535A41483F362D241B120900<127:0>
    );
    return FFmul_09<UInt(b)*8+:8>;
```

Library pseudocode for shared/functions/crypto/FFmul0B

```
// FFmul0B()
// =========

bits(8) FFmul0B(bits(8) b)
    bits(256*8) FFmul_0B = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0xA3A8B5BE8F849992FBF0EDE6D7DCC1CA<127:0> :
        /*E*/ 0x1318050E3F3429224B405D56676C717A<127:0> :
        /*D*/ 0xD8D3CEC5F4FFE2E9808B969DACA7BAB1<127:0> :
        /*C*/ 0x68637E75444F5259303B262D1C170A01<127:0> :
        /*B*/ 0x555E434879726F640D061B10212A373C<127:0> :
        /*A*/ 0xE5EEF3F8C9C2DFD4BDB6ABA0919A878C<127:0> :
        /*9*/ 0x2E2538330209141F767D606B5A514C47<127:0> :
        /*8*/ 0x9E958883B2B9A4AFC6CDD0DBEAE1FCF7<127:0> :
        /*7*/ 0x545F424978736E650C071A11202B363D<127:0> :
        /*6*/ 0xE4EFF2F9C8C3DED5BCB7AAA1909B868D<127:0> :
        /*5*/ 0x2F2439320308151E777C616A5B504D46<127:0> :
        /*4*/ 0x9F948982B3B8A5AEC7CCD1DAEBE0FDF6<127:0> :
        /*3*/ 0xA2A9B4BF8E859893FAF1ECE7D6DDC0CB<127:0> :
        /*2*/ 0x1219040F3E3528234A415C57666D707B<127:0> :
        /*1*/ 0xD9D2CFC4F5FEE3E8818A979CADA6BBB0<127:0> :
        /*0*/ 0x69627F74454E5358313A272C1D160B00<127:0>
    );
    return FFmul_0B<UInt(b)*8+:8>;
```

Library pseudocode for shared/functions/crypto/FFmul0D

```
// FFmul0D()
// =========

bits(8) FFmul0D(bits(8) b)
    bits(256*8) FFmul_0D = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x979A8D80A3AEB9B4FFF2E5E8CBC6D1DC<127:0> :
        /*E*/ 0x474A5D50737E69642F2235381B16010C<127:0> :
        /*D*/ 0x2C21363B1815020F44495E53707D6A67<127:0> :
        /*C*/ 0xFCF1E6EBC8C5D2DF94998E83A0ADBAB7<127:0> :
        /*B*/ 0xFAF7E0EDCEC3D4D9929F8885A6ABBCB1<127:0> :
        /*A*/ 0x2A27303D1E130409424F5855767B6C61<127:0> :
        /*9*/ 0x414C5B5675786F622924333E1D10070A<127:0> :
        /*8*/ 0x919C8B86A5A8BFB2F9F4E3EECDC0D7DA<127:0> :
        /*7*/ 0x4D40575A7974636E25283F32111C0B06<127:0> :
        /*6*/ 0x9D90878AA9A4B3BEF5F8EFE2C1CCDBD6<127:0> :
        /*5*/ 0xF6FBECE1C2CFD8D59E938489AAA7B0BD<127:0> :
        /*4*/ 0x262B3C31121F08054E4354597A77606D<127:0> :
        /*3*/ 0x202D3A3714190E034845525F7C71666B<127:0> :
        /*2*/ 0xF0FDEAE7C4C9DED39895828FACA1B6BB<127:0> :
        /*1*/ 0x9B96818CAFA2B5B8F3FEE9E4C7CADDD0<127:0> :
        /*0*/ 0x4B46515C7F726568232E3934171A0D00<127:0>
    );
    return FFmul_0D<UInt(b)*8+:8>;
```

Library pseudocode for shared/functions/crypto/FFmul0E

```
// FFmul0E()
// =========

bits(8) FFmul0E(bits(8) b)
    bits(256*8) FFmul_0E = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x8D83919FB5BBA9A7FDF3E1EFC5CBD9D7<127:0> :
        /*E*/ 0x6D63717F555B49471D13010F252B3937<127:0> :
        /*D*/ 0x56584A446E60727C26283A341E10020C<127:0> :
        /*C*/ 0xB6B8AAA48E80929CC6C8DAD4FEF0E2EC<127:0> :
        /*B*/ 0x202E3C321816040A505E4C426866747A<127:0> :
        /*A*/ 0xC0CEDCD2F8F6E4EAB0BEACA28886949A<127:0> :
        /*9*/ 0xFBF5E7E9C3CDDFD18B859799B3BDAFA1<127:0> :
        /*8*/ 0x1B150709232D3F316B657779535D4F41<127:0> :
        /*7*/ 0xCCC2D0DEF4FAE8E6BCB2A0AE848A9896<127:0> :
        /*6*/ 0x2C22303E141A08065C52404E646A7876<127:0> :
        /*5*/ 0x17190B052F21333D67697B755F51434D<127:0> :
        /*4*/ 0xF7F9EBE5CFC1D3DD87899B95BFB1A3AD<127:0> :
        /*3*/ 0x616F7D735957454B111F0D032927353B<127:0> :
        /*2*/ 0x818F9D93B9B7A5ABF1FFEDE3C9C7D5DB<127:0> :
        /*1*/ 0xBAB4A6A8828C9E90CAC4D6D8F2FCEEE0<127:0> :
        /*0*/ 0x5A544648626C7E702A243638121C0E00<127:0>
    );
    return FFmul_0E<UInt(b)*8+:8>;
```

Library pseudocode for shared/functions/crypto/HaveAESExt

```
// HaveAESExt()
// ============
// TRUE if AES cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveAESExt()
    return boolean IMPLEMENTATION_DEFINED "Has AES Crypto instructions";
```

Library pseudocode for shared/functions/crypto/HaveBit128PMULLExt

```
// HaveBit128PMULLExt()
// ====================
// TRUE if 128 bit form of PMULL instructions support is implemented,
// FALSE otherwise.

boolean HaveBit128PMULLExt()
    return boolean IMPLEMENTATION_DEFINED "Has 128-bit form of PMULL instructions";
```

Library pseudocode for shared/functions/crypto/HaveSHA1Ext

```
// HaveSHA1Ext()
// =============
// TRUE if SHA1 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA1Ext()
    return boolean IMPLEMENTATION_DEFINED "Has SHA1 Crypto instructions";
```

Library pseudocode for shared/functions/crypto/HaveSHA256Ext

```
// HaveSHA256Ext()
// ===============
// TRUE if SHA256 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA256Ext()
    return boolean IMPLEMENTATION_DEFINED "Has SHA256 Crypto instructions";
```

Library pseudocode for shared/functions/crypto/HaveSHA3Ext

```
// HaveSHA3Ext()
// =============
// TRUE if SHA3 cryptographic instructions support is implemented,
// and when SHA1 and SHA2 basic cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA3Ext()
    if !HasArchVersion(ARMv8p2) || !(HaveSHA1Ext() && HaveSHA256Ext()) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SHA3 Crypto instructions";
```

Library pseudocode for shared/functions/crypto/HaveSHA512Ext

```
// HaveSHA512Ext()
// ===============
// TRUE if SHA512 cryptographic instructions support is implemented,
// and when SHA1 and SHA2 basic cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA512Ext()
    if !HasArchVersion(ARMv8p2) || !(HaveSHA1Ext() && HaveSHA256Ext()) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SHA512 Crypto instructions";
```

Library pseudocode for shared/functions/crypto/HaveSM3Ext

```
// HaveSM3Ext()
// ============
// TRUE if SM3 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSM3Ext()
    if !HasArchVersion(ARMv8p2) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SM3 Crypto instructions";
```

Library pseudocode for shared/functions/crypto/HaveSM4Ext

```
// HaveSM4Ext()
// ============
// TRUE if SM4 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSM4Ext()
    if !HasArchVersion(ARMv8p2) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SM4 Crypto instructions";
```

Library pseudocode for shared/functions/crypto/ROL

```
// ROL()
// =====

bits(N) ROL(bits(N) x, integer shift)
    assert shift >= 0 && shift <= N;
    if (shift == 0) then
        return x;
    return ROR(x, N-shift);
```

Library pseudocode for shared/functions/crypto/SHA256hash

```
// SHA256hash()
// ============

bits(128) SHA256hash(bits (128) x_in, bits(128) y_in, bits(128) w, boolean part1)
    bits(32) chs, maj, t;
    bits(128) x = x_in;
    bits(128) y = y_in;

    for e = 0 to 3
        chs = SHAchoose(y<31:0>, y<63:32>, y<95:64>);
        maj = SHAmajority(x<31:0>, x<63:32>, x<95:64>);
        t = y<127:96> + SHAhashSIGMA1(y<31:0>) + chs + Elem[w, e, 32];
        x<127:96> = t + x<127:96>;
        y<127:96> = t + SHAhashSIGMA0(x<31:0>) + maj;
        <y, x> = ROL(y : x, 32);
    return (if part1 then x else y);
```

Library pseudocode for shared/functions/crypto/SHAchoose

```
// SHAchoose()
// ===========

bits(32) SHAchoose(bits(32) x, bits(32) y, bits(32) z)
    return (((y EOR z) AND x) EOR z);
```

Library pseudocode for shared/functions/crypto/SHAhashSIGMA0

```
// SHAhashSIGMA0()
// ===============

bits(32) SHAhashSIGMA0(bits(32) x)
    return ROR(x, 2) EOR ROR(x, 13) EOR ROR(x, 22);
```

Library pseudocode for shared/functions/crypto/SHAhashSIGMA1

```
// SHAhashSIGMA1()
// ===============

bits(32) SHAhashSIGMA1(bits(32) x)
    return ROR(x, 6) EOR ROR(x, 11) EOR ROR(x, 25);
```

Library pseudocode for shared/functions/crypto/SHAmajority

```
// SHAmajority()
// =============

bits(32) SHAmajority(bits(32) x, bits(32) y, bits(32) z)
    return ((x AND y) OR ((x OR y) AND z));
```

Library pseudocode for shared/functions/crypto/SHAparity

```
// SHAparity()
// ===========

bits(32) SHAparity(bits(32) x, bits(32) y, bits(32) z)
    return (x EOR y EOR z);
```

Library pseudocode for shared/functions/crypto/Sbox

```
// Sbox()
// ======
// Used in SM4E crypto instruction

bits(8) Sbox(bits(8) sboxin)
    bits(8) sboxout;
    bits(2048) sboxstring = 0xd690e9fecce13db716b614c228fb2c052b679a762abe04c3aa441326498606999c4250f491e

    sboxout = sboxstring<(255-UInt(sboxin))*8+7:(255-UInt(sboxin))*8>;
    return sboxout;
```

Library pseudocode for shared/functions/exclusive/ClearExclusiveByAddress

```
// Clear the global Exclusives monitors for all PEs EXCEPT processorid if they
// record any part of the physical address region of size bytes starting at paddress.
// It is IMPLEMENTATION DEFINED whether the global Exclusives monitor for processorid
// is also cleared if it records any part of the address region.
ClearExclusiveByAddress(FullAddress paddress, integer processorid, integer size);
```

Library pseudocode for shared/functions/exclusive/ClearExclusiveLocal

```
// Clear the local Exclusives monitor for the specified processorid.
ClearExclusiveLocal(integer processorid);
```

Library pseudocode for shared/functions/exclusive/ClearExclusiveMonitors

```
// ClearExclusiveMonitors()
// ========================
// Clear the local Exclusives monitor for the executing PE.

ClearExclusiveMonitors()
    ClearExclusiveLocal(ProcessorID());
```

Library pseudocode for shared/functions/exclusive/ExclusiveMonitorsStatus

```
// Returns '0' to indicate success if the last memory write by this PE was to
// the same physical address region endorsed by ExclusiveMonitorsPass().
// Returns '1' to indicate failure if address translation resulted in a different
// physical address.
bit ExclusiveMonitorsStatus();
```

Library pseudocode for shared/functions/exclusive/IsExclusiveGlobal

```
// Return TRUE if the global Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.
boolean IsExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
```

Library pseudocode for shared/functions/exclusive/IsExclusiveLocal

```
// Return TRUE if the local Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.
boolean IsExclusiveLocal(FullAddress paddress, integer processorid, integer size);
```

Library pseudocode for shared/functions/exclusive/MarkExclusiveGlobal

```
// Record the physical address region of size bytes starting at paddress in
// the global Exclusives monitor for processorid.
MarkExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
```

Library pseudocode for shared/functions/exclusive/MarkExclusiveLocal

```
// Record the physical address region of size bytes starting at paddress in
// the local Exclusives monitor for processorid.
MarkExclusiveLocal(FullAddress paddress, integer processorid, integer size);
```

Library pseudocode for shared/functions/exclusive/ProcessorID

```
// Return the ID of the currently executing PE.
integer ProcessorID();
```

Library pseudocode for shared/functions/extension/AArch32.HaveHPDExt

```
// AArch32.HaveHPDExt()
// ====================

boolean AArch32.HaveHPDExt()
    return (HasArchVersion(ARMv8p2) &&
            boolean IMPLEMENTATION_DEFINED "Has AArch32 hierarchical permission disables");
```

Library pseudocode for shared/functions/extension/AArch64.HaveHPDExt

```
// AArch64.HaveHPDExt()
// ====================

boolean AArch64.HaveHPDExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/extension/Have16bitVMID

```
// Have16bitVMID()
// ===============
// Returns TRUE if EL2 and support for a 16-bit VMID are implemented.

boolean Have16bitVMID()
    return (HasArchVersion(ARMv8p1) && HaveEL(EL2) &&
        boolean IMPLEMENTATION_DEFINED "Has 16-bit VMID");
```

Library pseudocode for shared/functions/extension/Have52BitIPAAndPASpaceExt

```
// Have52BitIPAAndPASpaceExt()
// ===========================
// Returns TRUE if 52-bit IPA and PA extension support
// is implemented, and FALSE otherwise.

boolean Have52BitIPAAndPASpaceExt()
    return (HasArchVersion(ARMv8p7) &&
            boolean IMPLEMENTATION_DEFINED "Has 52-bit IPA and PA support" &&
            Have52BitVAExt() && Have52BitPAExt());
```

Library pseudocode for shared/functions/extension/Have52BitPAExt

```
// Have52BitPAExt()
// ================
// Returns TRUE if Large Physical Address extension
// support is implemented and FALSE otherwise.

boolean Have52BitPAExt()
    return (HasArchVersion(ARMv8p2) &&
           boolean IMPLEMENTATION_DEFINED "Has large 52-bit PA/IPA support");
```

Library pseudocode for shared/functions/extension/Have52BitVAExt

```
// Have52BitVAExt()
// ================
// Returns TRUE if Large Virtual Address extension
// support is implemented and FALSE otherwise.

boolean Have52BitVAExt()
    return (HasArchVersion(ARMv8p2) &&
            boolean IMPLEMENTATION_DEFINED "Has large 52-bit VA support");
```

Library pseudocode for shared/functions/extension/HaveAArch32BF16Ext

```
// HaveAArch32BF16Ext()
// ====================
// Returns TRUE if AArch32 BFloat16 instruction support is implemented, and FALSE otherwise.

boolean HaveAArch32BF16Ext()
    return (HasArchVersion(ARMv8p2) &&
            boolean IMPLEMENTATION_DEFINED "Has AArch32 BFloat16 extension");
```

Library pseudocode for shared/functions/extension/HaveAArch32Int8MatMulExt

```
// HaveAArch32Int8MatMulExt()
// ==========================
// Returns TRUE if AArch32 8-bit integer matrix multiply instruction support
// implemented, and FALSE otherwise.

boolean HaveAArch32Int8MatMulExt()
    return (HasArchVersion(ARMv8p2) &&
            boolean IMPLEMENTATION_DEFINED "Has AArch32 Int8 Mat Mul extension");
```

Library pseudocode for shared/functions/extension/HaveAccessFlagUpdateExt

```
// HaveAccessFlagUpdateExt()
// =========================

boolean HaveAccessFlagUpdateExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/extension/HaveAltFP

```
// HaveAltFP()
// ===========
// Returns TRUE if alternative Floating-point extension support
// is implemented, and FALSE otherwise.

boolean HaveAltFP()
    return HasArchVersion(ARMv8p7);
```

Library pseudocode for shared/functions/extension/HaveAtomicExt

```
// HaveAtomicExt()
// ===============

boolean HaveAtomicExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/extension/HaveBF16Ext

```
// HaveBF16Ext()
// =============
// Returns TRUE if AArch64 BFloat16 instruction support is implemented, and FALSE otherwise.

boolean HaveBF16Ext()
    return (HasArchVersion(ARMv8p6) ||
            (HasArchVersion(ARMv8p2) &&
             boolean IMPLEMENTATION_DEFINED "Has AArch64 BFloat16 extension"));
```

Library pseudocode for shared/functions/extension/HaveBRBEv1p1

```
// HaveBRBEv1p1()
// ==============
// Returns TRUE if BRBEv1p1 extension is implemented, and FALSE otherwise.

boolean HaveBRBEv1p1()
    return (HasArchVersion(ARMv9p3) &&
        boolean IMPLEMENTATION_DEFINED "Has BRBEv1p1 extension");
```

Library pseudocode for shared/functions/extension/HaveBRBExt

```
// HaveBRBExt()
// ============
// Returns TRUE if Branch Record Buffer Extension is implemented, and FALSE otherwise.

boolean HaveBRBExt()
    return boolean IMPLEMENTATION_DEFINED "Has Branch Record Buffer Extension";
```

Library pseudocode for shared/functions/extension/HaveBTIExt

```
// HaveBTIExt()
// ============
// Returns TRUE if support for Branch Target Indentification is implemented.

boolean HaveBTIExt()
    return HasArchVersion(ARMv8p5);
```

Library pseudocode for shared/functions/extension/HaveBlockBBM

```
// HaveBlockBBM()
// ==============
// Returns TRUE if support for changing block size without requiring
// break-before-make is implemented.

boolean HaveBlockBBM()
    return HasArchVersion(ARMv8p4);
```

Library pseudocode for shared/functions/extension/HaveCNTSCExt

```
// HaveCNTSCExt()
// ==============
// Returns TRUE if the Generic Counter Scaling is implemented, and FALSE
// otherwise.

boolean HaveCNTSCExt()
    return (HasArchVersion(ARMv8p4) &&
            boolean IMPLEMENTATION_DEFINED "Has Generic Counter Scaling support");
```

Library pseudocode for shared/functions/extension/HaveCommonNotPrivateTransExt

```
// HaveCommonNotPrivateTransExt()
// ==============================

boolean HaveCommonNotPrivateTransExt()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveDGHExt

```
// HaveDGHExt()
// ============
// Returns TRUE if Data Gathering Hint instruction support is implemented, and
// FALSE otherwise.

boolean HaveDGHExt()
    return boolean IMPLEMENTATION_DEFINED "Has AArch64 DGH extension";
```

Library pseudocode for shared/functions/extension/HaveDITExt

```
// HaveDITExt()
// ============

boolean HaveDITExt()
    return HasArchVersion(ARMv8p4);
```

Library pseudocode for shared/functions/extension/HaveDOTPExt

```
// HaveDOTPExt()
// =============
// Returns TRUE if Dot Product feature support is implemented, and FALSE otherwise.

boolean HaveDOTPExt()
    return (HasArchVersion(ARMv8p4) ||
            (HasArchVersion(ARMv8p2) &&
            boolean IMPLEMENTATION_DEFINED "Has Dot Product extension"));
```

Library pseudocode for shared/functions/extension/HaveDirtyBitModifierExt

```
// HaveDirtyBitModifierExt()
// =========================

boolean HaveDirtyBitModifierExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/extension/HaveDoPD

```
// HaveDoPD()
// ==========
// Returns TRUE if Debug Over Power Down extension
// support is implemented and FALSE otherwise.

boolean HaveDoPD()
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has DoPD extension";
```

Library pseudocode for shared/functions/extension/HaveDoubleFaultExt

```
// HaveDoubleFaultExt()
// ====================

boolean HaveDoubleFaultExt()
    return (HasArchVersion(ARMv8p4) && HaveEL(EL3) && !ELUsingAArch32(EL3) && HaveIESB());
```

Library pseudocode for shared/functions/extension/HaveDoubleLock

```
// HaveDoubleLock()
// ================
// Returns TRUE if support for the OS Double Lock is implemented.

boolean HaveDoubleLock()
    return (!HasArchVersion(ARMv8p4) ||
            boolean IMPLEMENTATION_DEFINED "OS Double Lock is implemented");
```

Library pseudocode for shared/functions/extension/HaveE0PDExt

```
// HaveE0PDExt()
// =============
// Returns TRUE if support for constant fault times for unprivileged accesses
// to the memory map is implemented.

boolean HaveE0PDExt()
    return HasArchVersion(ARMv8p5);
```

Library pseudocode for shared/functions/extension/HaveECVExt

```
// HaveECVExt()
// ============
// Returns TRUE if Enhanced Counter Virtualization extension
// support is implemented, and FALSE otherwise.

boolean HaveECVExt()
    return HasArchVersion(ARMv8p6);
```

Library pseudocode for shared/functions/extension/HaveETExt

```
// HaveETExt()
// ===========
// Returns TRUE if Embedded Trace Extension is implemented, and FALSE otherwise.

boolean HaveETExt()
    return boolean IMPLEMENTATION_DEFINED "Has Embedded Trace Extension";
```

Library pseudocode for shared/functions/extension/HaveExtendedCacheSets

```
// HaveExtendedCacheSets()
// =======================

boolean HaveExtendedCacheSets()
    return HasArchVersion(ARMv8p3);
```

Library pseudocode for shared/functions/extension/HaveExtendedECDebugEvents

```
// HaveExtendedECDebugEvents()
// ===========================

boolean HaveExtendedECDebugEvents()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveExtendedExecuteNeverExt

```
// HaveExtendedExecuteNeverExt()
// =============================

boolean HaveExtendedExecuteNeverExt()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveFCADDExt

```
// HaveFCADDExt()
// ==============

boolean HaveFCADDExt()
    return HasArchVersion(ARMv8p3);
```

Library pseudocode for shared/functions/extension/HaveFGTExt

```
// HaveFGTExt()
// ============
// Returns TRUE if Fine Grained Trap is implemented, and FALSE otherwise.

boolean HaveFGTExt()
    return HasArchVersion(ARMv8p6);
```

Library pseudocode for shared/functions/extension/HaveFJCVTZSExt

```
// HaveFJCVTZSExt()
// ================

boolean HaveFJCVTZSExt()
    return HasArchVersion(ARMv8p3);
```

Library pseudocode for shared/functions/extension/HaveFP16MulNoRoundingToFP32Ext

```
// HaveFP16MulNoRoundingToFP32Ext()
// ================================
// Returns TRUE if has FP16 multiply with no intermediate rounding accumulate
// to FP32 instructions, and FALSE otherwise

boolean HaveFP16MulNoRoundingToFP32Ext()
    if !HaveFP16Ext() then return FALSE;
    if HasArchVersion(ARMv8p4) then return TRUE;
    return (HasArchVersion(ARMv8p2) &&
            boolean IMPLEMENTATION_DEFINED "Has accumulate FP16 product into FP32 extension");
```

Library pseudocode for shared/functions/extension/HaveFeatCMOW

```
// HaveFeatCMOW()
// ==============
// Returns TRUE if the SCTLR_EL1.CMOW bit is implemented and the SCTLR_EL2.CMOW and
// HCRX_EL2.CMOW bits are implemented if EL2 is implemented.

boolean HaveFeatCMOW()
    return HasArchVersion(ARMv8p8);
```

Library pseudocode for shared/functions/extension/HaveFeatHBC

```
// HaveFeatHBC()
// =============
// Returns TRUE if the BC instruction is implemented, and FALSE otherwise.

boolean HaveFeatHBC()
    return HasArchVersion(ARMv8p8);
```

Library pseudocode for shared/functions/extension/HaveFeatHCX

```
// HaveFeatHCX()
// =============
// Returns TRUE if HCRX_EL2 Trap Control register is implemented,
// and FALSE otherwise.

boolean HaveFeatHCX()
    return HasArchVersion(ARMv8p7);
```

Library pseudocode for shared/functions/extension/HaveFeatHPMN0

```
// HaveFeatHPMN0()
// ===============
// Returns TRUE if HDCR.HPMN or MDCR_EL2.HPMN is permitted to be 0 without
// generating UNPREDICTABLE behavior, and FALSE otherwise.

boolean HaveFeatHPMN0()
    return HasArchVersion(ARMv8p6) && HavePMUv3() && HaveFGTExt() && HaveEL(EL2);
```

Library pseudocode for shared/functions/extension/HaveFeatLS64

```
// HaveFeatLS64()
// ==============
// Returns TRUE if the LD64B, ST64B instructions are
// supported, and FALSE otherwise.

boolean HaveFeatLS64()
    return (HasArchVersion(ARMv8p7) &&
            boolean IMPLEMENTATION_DEFINED "Has Load Store 64-Byte instruction support");
```

Library pseudocode for shared/functions/extension/HaveFeatLS64_ACCDATA

```
// HaveFeatLS64_ACCDATA()
// ======================
// Returns TRUE if the ST64BV0 instruction is
// supported, and FALSE otherwise.

boolean HaveFeatLS64_ACCDATA()
    return (HasArchVersion(ARMv8p7) && HaveFeatLS64_V() &&
            boolean IMPLEMENTATION_DEFINED "Has Store 64-Byte EL0 with return instruction support");
```

Library pseudocode for shared/functions/extension/HaveFeatLS64_V

```
// HaveFeatLS64_V()
// ================
// Returns TRUE if the ST64BV instruction is
// supported, and FALSE otherwise.

boolean HaveFeatLS64_V()
    return (HasArchVersion(ARMv8p7) && HaveFeatLS64() &&
            boolean IMPLEMENTATION_DEFINED "Has Store 64-Byte with return instruction support");
```

Library pseudocode for shared/functions/extension/HaveFeatMOPS

```
// HaveFeatMOPS()
// ==============
// Returns TRUE if the CPY* and SET* instructions are supported, and FALSE otherwise.

boolean HaveFeatMOPS()
    return HasArchVersion(ARMv8p8);
```

Library pseudocode for shared/functions/extension/HaveFeatNMI

```
// HaveFeatNMI()
// =============
// Returns TRUE if the Non-Maskable Interrupt extension is
// implemented, and FALSE otherwise.

boolean HaveFeatNMI()
    return HasArchVersion(ARMv8p8);
```

Library pseudocode for shared/functions/extension/HaveFeatRPRES

```
// HaveFeatRPRES()
// ===============
// Returns TRUE if reciprocal estimate implements 12-bit precision
// when FPCR.AH=1, and FALSE otherwise.

boolean HaveFeatRPRES()
    return (HasArchVersion(ARMv8p7) &&
        (boolean IMPLEMENTATION_DEFINED "Has increased Reciprocal Estimate and Square Root Estimate preci
        HaveAltFP());
```

Library pseudocode for shared/functions/extension/HaveFeatTIDCP1

```
// HaveFeatTIDCP1()
// ================
// Returns TRUE if the SCTLR_EL1.TIDCP bit is implemented and the SCTLR_EL2.TIDCP bit
// is implemented if EL2 is implemented.

boolean HaveFeatTIDCP1()
    return HasArchVersion(ARMv8p8);
```

Library pseudocode for shared/functions/extension/HaveFeatWFxT

```
// HaveFeatWFxT()
// ==============
// Returns TRUE if WFET and WFIT instruction support is implemented,
// and FALSE otherwise.

boolean HaveFeatWFxT()
    return HasArchVersion(ARMv8p7);
```

Library pseudocode for shared/functions/extension/HaveFeatXS

```
// HaveFeatXS()
// ============
// Returns TRUE if XS attribute and the TLBI and DSB instructions with nXS qualifier
// are supported, and FALSE otherwise.

boolean HaveFeatXS()
    return HasArchVersion(ARMv8p7);
```

Library pseudocode for shared/functions/extension/HaveFlagFormatExt

```
// HaveFlagFormatExt()
// ===================
// Returns TRUE if flag format conversion instructions implemented.

boolean HaveFlagFormatExt()
    return HasArchVersion(ARMv8p5);
```

Library pseudocode for shared/functions/extension/HaveFlagManipulateExt

```
// HaveFlagManipulateExt()
// =======================
// Returns TRUE if flag manipulate instructions are implemented.

boolean HaveFlagManipulateExt()
    return HasArchVersion(ARMv8p4);
```

Library pseudocode for shared/functions/extension/HaveFrintExt

```
// HaveFrintExt()
// ==============
// Returns TRUE if FRINT instructions are implemented.

boolean HaveFrintExt()
    return HasArchVersion(ARMv8p5);
```

Library pseudocode for shared/functions/extension/HaveGTGExt

```
// HaveGTGExt()
// ============
// Returns TRUE if support for guest translation granule size is implemented.

boolean HaveGTGExt()
    return HasArchVersion(ARMv8p5);
```

Library pseudocode for shared/functions/extension/HaveHPMDExt

```
// HaveHPMDExt()
// =============

boolean HaveHPMDExt()
    return HavePMUv3p1();
```

Library pseudocode for shared/functions/extension/HaveIDSExt

```
// HaveIDSExt()
// ============
// Returns TRUE if ID register handling feature is implemented.

boolean HaveIDSExt()
    return HasArchVersion(ARMv8p4);
```

Library pseudocode for shared/functions/extension/HaveIESB

```
// HaveIESB()
// ==========

boolean HaveIESB()
    return (HaveRASExt() &&
            boolean IMPLEMENTATION_DEFINED "Has Implicit Error Synchronization Barrier");
```

Library pseudocode for shared/functions/extension/HaveInt8MatMulExt

```
// HaveInt8MatMulExt()
// ===================
// Returns TRUE if AArch64 8-bit integer matrix multiply instruction support
// implemented, and FALSE otherwise.

boolean HaveInt8MatMulExt()
    return (HasArchVersion(ARMv8p6) ||
            (HasArchVersion(ARMv8p2) &&
             boolean IMPLEMENTATION_DEFINED "Has AArch64 Int8 Mat Mul extension"));
```

Library pseudocode for shared/functions/extension/HaveLSE2Ext

```
// HaveLSE2Ext()
// =============
// Returns TRUE if LSE2 is implemented, and FALSE otherwise.

boolean HaveLSE2Ext()
    return HasArchVersion(ARMv8p4);
```

Library pseudocode for shared/functions/extension/HaveMPAMExt

```
// HaveMPAMExt()
// =============
// Returns TRUE if MPAM is implemented, and FALSE otherwise.

boolean HaveMPAMExt()
    return (HasArchVersion(ARMv8p2) &&
            boolean IMPLEMENTATION_DEFINED "Has MPAM extension");
```

Library pseudocode for shared/functions/extension/HaveMPAMv0p1Ext

```
// HaveMPAMv0p1Ext()
// =================
// Returns TRUE if MPAMv0p1 is implemented, and FALSE otherwise.

boolean HaveMPAMv0p1Ext()
    return (HasArchVersion(ARMv8p6) &&
            HaveMPAMExt() &&
            boolean IMPLEMENTATION_DEFINED "Has enhanced MPAMv0p1 extension");
```

Library pseudocode for shared/functions/extension/HaveMPAMv1p1Ext

```
// HaveMPAMv1p1Ext()
// =================
// Returns TRUE if MPAMv1p1 is implemented, and FALSE otherwise.

boolean HaveMPAMv1p1Ext()
    return (HasArchVersion(ARMv8p6) &&
            HaveMPAMExt() &&
            boolean IMPLEMENTATION_DEFINED "Has enhanced MPAMv1p1 extension");
```

Library pseudocode for shared/functions/extension/HaveMTE2Ext

```
// HaveMTE2Ext()
// =============
// Returns TRUE if MTE support is beyond EL0, and FALSE otherwise.

boolean HaveMTE2Ext()
    if !HasArchVersion(ARMv8p5) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has MTE2 extension";
```

Library pseudocode for shared/functions/extension/HaveMTE3Ext

```
// HaveMTE3Ext()
// =============
// Returns TRUE if MTE Asymmetric Fault Handling support is
// implemented, and FALSE otherwise.

boolean HaveMTE3Ext()
    return ((HasArchVersion(ARMv8p7) && HaveMTE2Ext()) || (HasArchVersion(ARMv8p5) &&
            boolean IMPLEMENTATION_DEFINED "Has MTE3 extension"));
```

Library pseudocode for shared/functions/extension/HaveMTEExt

```
// HaveMTEExt()
// ============
// Returns TRUE if instruction-only MTE implemented, and FALSE otherwise.

boolean HaveMTEExt()
    if !HasArchVersion(ARMv8p5) then
        return FALSE;
    if HaveMTE2Ext() then
        return TRUE;
    return boolean IMPLEMENTATION_DEFINED "Has MTE extension";
```

Library pseudocode for shared/functions/extension/HaveNV2Ext

```
// HaveNV2Ext()
// ============
// Returns TRUE if Enhanced Nested Virtualization is implemented.

boolean HaveNV2Ext()
    return (HasArchVersion(ARMv8p4) && HaveNVExt()
            && boolean IMPLEMENTATION_DEFINED "Has support for Enhanced Nested Virtualization");
```

Library pseudocode for shared/functions/extension/HaveNVExt

```
// HaveNVExt()
// ===========
// Returns TRUE if Nested Virtualization is implemented.

boolean HaveNVExt()
    return (HasArchVersion(ARMv8p3) &&
            boolean IMPLEMENTATION_DEFINED "Has Nested Virtualization");
```

Library pseudocode for shared/functions/extension/HaveNoSecurePMUDisableOverride

```
// HaveNoSecurePMUDisableOverride()
// ================================

boolean HaveNoSecurePMUDisableOverride()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveNoninvasiveDebugAuth

```
// HaveNoninvasiveDebugAuth()
// ==========================
// Returns TRUE if the Non-invasive debug controls are implemented.

boolean HaveNoninvasiveDebugAuth()
    return !HasArchVersion(ARMv8p4);
```

Library pseudocode for shared/functions/extension/HavePAN3Ext

```
// HavePAN3Ext()
// =============
// Returns TRUE if SCTLR_EL1.EPAN and SCTLR_EL2.EPAN support is implemented,
// and FALSE otherwise.

boolean HavePAN3Ext()
    return HasArchVersion(ARMv8p7) || (HasArchVersion(ARMv8p1) &&
           boolean IMPLEMENTATION_DEFINED "Has PAN3 extension");
```

Library pseudocode for shared/functions/extension/HavePANExt

```
// HavePANExt()
// ============

boolean HavePANExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/extension/HavePMUv3

```
// HavePMUv3()
// ===========
// Returns TRUE if the Performance Monitors extension is implemented, and FALSE otherwise.

boolean HavePMUv3()
    return boolean IMPLEMENTATION_DEFINED "Has Performance Monitors extension";
```

Library pseudocode for shared/functions/extension/HavePMUv3TH

```
// HavePMUv3TH()
// =============
// Returns TRUE if the PMUv3 threshold extension is implemented, and FALSE otherwise.

boolean HavePMUv3TH()
    return (HasArchVersion(ARMv8p8) && HavePMUv3() &&
        boolean IMPLEMENTATION_DEFINED "Has PMUv3 threshold extension");
```

Library pseudocode for shared/functions/extension/HavePMUv3p1

```
// HavePMUv3p1()
// =============
// Returns TRUE if the Performance Monitors extension is implemented, and FALSE otherwise.

boolean HavePMUv3p1()
    return HasArchVersion(ARMv8p1) && HavePMUv3();
```

Library pseudocode for shared/functions/extension/HavePMUv3p4

```
// HavePMUv3p4()
// =============
// Returns TRUE if the PMUv3.4 extension is implemented, and FALSE otherwise.

boolean HavePMUv3p4()
    return HasArchVersion(ARMv8p4) && HavePMUv3();
```

Library pseudocode for shared/functions/extension/HavePMUv3p5

```
// HavePMUv3p5()
// =============
// Returns TRUE if the PMUv3.5 extension is implemented, and FALSE otherwise.

boolean HavePMUv3p5()
    return HasArchVersion(ARMv8p5) && HavePMUv3();
```

Library pseudocode for shared/functions/extension/HavePMUv3p7

```
// HavePMUv3p7()
// =============
// Returns TRUE if the PMUv3.7 extension is implemented, and FALSE otherwise.

boolean HavePMUv3p7()
    return HasArchVersion(ARMv8p7) && HavePMUv3();
```

Library pseudocode for shared/functions/extension/HavePageBasedHardwareAttributes

```
// HavePageBasedHardwareAttributes()
// =================================

boolean HavePageBasedHardwareAttributes()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HavePrivATExt

```
// HavePrivATExt()
// ===============

boolean HavePrivATExt()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveQRDMLAHExt

```
// HaveQRDMLAHExt()
// ================

boolean HaveQRDMLAHExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/extension/HaveRASExt

```
// HaveRASExt()
// ============

boolean HaveRASExt()
    return (HasArchVersion(ARMv8p2) ||
            boolean IMPLEMENTATION_DEFINED "Has RAS extension");
```

Library pseudocode for shared/functions/extension/HaveRME

```
// HaveRME()
// =========
// Returns TRUE if the Realm Management Extension is implemented, and FALSE
// otherwise.

boolean HaveRME()
    return boolean IMPLEMENTATION_DEFINED "Has RME extension";
```

Library pseudocode for shared/functions/extension/HaveRNG

```
// HaveRNG()
// =========
// Returns TRUE if Random Number Generator extension
// support is implemented and FALSE otherwise.

boolean HaveRNG()
    return HasArchVersion(ARMv8p5) && boolean IMPLEMENTATION_DEFINED "Has RNG extension";
```

Library pseudocode for shared/functions/extension/HaveSBExt

```
// HaveSBExt()
// ===========
// Returns TRUE if support for SB is implemented, and FALSE otherwise.

boolean HaveSBExt()
    return HasArchVersion(ARMv8p5) || boolean IMPLEMENTATION_DEFINED "Has SB extension";
```

Library pseudocode for shared/functions/extension/HaveSSBSExt

```
// HaveSSBSExt()
// =============
// Returns TRUE if support for SSBS is implemented, and FALSE otherwise.

boolean HaveSSBSExt()
    return HasArchVersion(ARMv8p5) || boolean IMPLEMENTATION_DEFINED "Has SSBS extension";
```

Library pseudocode for shared/functions/extension/HaveSecureEL2Ext

```
// HaveSecureEL2Ext()
// ==================
// Returns TRUE if Secure EL2 is implemented.

boolean HaveSecureEL2Ext()
    return HasArchVersion(ARMv8p4);
```

Library pseudocode for shared/functions/extension/HaveSecureExtDebugView

```
// HaveSecureExtDebugView()
// ========================
// Returns TRUE if support for Secure and Non-secure views of debug peripherals
// is implemented.

boolean HaveSecureExtDebugView()
    return HasArchVersion(ARMv8p4);
```

Library pseudocode for shared/functions/extension/HaveSelfHostedTrace

```
// HaveSelfHostedTrace()
// =====================

boolean HaveSelfHostedTrace()
    return HasArchVersion(ARMv8p4);
```

Library pseudocode for shared/functions/extension/HaveSmallTranslationTblExt

```
// HaveSmallTranslationTblExt()
// ============================
// Returns TRUE if Small Translation Table Support is implemented.

boolean HaveSmallTranslationTableExt()
    return (HasArchVersion(ARMv8p4) &&
            boolean IMPLEMENTATION_DEFINED "Has Small Translation Table extension");
```

Library pseudocode for shared/functions/extension/HaveSoftwareLock

```
// HaveSoftwareLock()
// ==================
// Returns TRUE if Software Lock is implemented.

boolean HaveSoftwareLock(Component component)
    if Havev8p4Debug() then
        return FALSE;
    if HaveDoPD() && component != Component_CTI then
        return FALSE;
    case component of
        when Component_Debug
            return boolean IMPLEMENTATION_DEFINED "Debug has Software Lock";
        when Component_PMU
            return boolean IMPLEMENTATION_DEFINED "PMU has Software Lock";
        when Component_CTI
            return boolean IMPLEMENTATION_DEFINED "CTI has Software Lock";
        otherwise
            Unreachable();
```

Library pseudocode for shared/functions/extension/HaveStage2MemAttrControl

```
// HaveStage2MemAttrControl()
// ==========================
// Returns TRUE if support for Stage2 control of memory types and cacheability
// attributes is implemented.

boolean HaveStage2MemAttrControl()
    return HasArchVersion(ARMv8p4);
```

Library pseudocode for shared/functions/extension/HaveStatisticalProfiling

```
// HaveStatisticalProfiling()
// ==========================
// Returns TRUE if Statistical Profiling Extension is implemented,
// and FALSE otherwise.

boolean HaveStatisticalProfiling()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveStatisticalProfilingv1p1

```
// HaveStatisticalProfilingv1p1()
// ==============================
// Returns TRUE if the SPEv1p1 extension is implemented, and FALSE otherwise.

boolean HaveStatisticalProfilingv1p1()
    return (HasArchVersion(ARMv8p3) &&
        boolean IMPLEMENTATION_DEFINED "Has SPEv1p1 extension");
```

Library pseudocode for shared/functions/extension/HaveStatisticalProfilingv1p2

```
// HaveStatisticalProfilingv1p2()
// ==============================
// Returns TRUE if the SPEv1p2 extension is implemented, and FALSE otherwise.

boolean HaveStatisticalProfilingv1p2()
    return (HasArchVersion(ARMv8p7) && HaveStatisticalProfiling() &&
        boolean IMPLEMENTATION_DEFINED "Has SPEv1p2 extension");
```

Library pseudocode for shared/functions/extension/HaveTME

```
// HaveTME()
// =========

boolean HaveTME()
    return boolean IMPLEMENTATION_DEFINED "Has Transactional Memory extension";
```

Library pseudocode for shared/functions/extension/HaveTWEDExt

```
// HaveTWEDExt()
// =============
// Returns TRUE if Delayed Trapping of WFE instruction support is implemented,
// and FALSE otherwise.

boolean HaveTWEDExt()
    return boolean IMPLEMENTATION_DEFINED "Has TWED extension";
```

Library pseudocode for shared/functions/extension/HaveTraceBufferExtension

```
// HaveTraceBufferExtension()
// ==========================
// Returns TRUE if Trace Buffer Extension is implemented, and FALSE otherwise.

boolean HaveTraceBufferExtension()
    return boolean IMPLEMENTATION_DEFINED "Trace Buffer Extension implemented";
```

Library pseudocode for shared/functions/extension/HaveTraceExt

```
// HaveTraceExt()
// ==============
// Returns TRUE if Trace functionality as described by the Trace Architecture
// is implemented.

boolean HaveTraceExt()
    return boolean IMPLEMENTATION_DEFINED "Has Trace Architecture functionality";
```

Library pseudocode for shared/functions/extension/HaveTrapLoadStoreMultipleDeviceExt

```
// HaveTrapLoadStoreMultipleDeviceExt()
// ====================================

boolean HaveTrapLoadStoreMultipleDeviceExt()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveUAOExt

```
// HaveUAOExt()
// ============

boolean HaveUAOExt()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveV82Debug

```
// HaveV82Debug()
// ==============

boolean HaveV82Debug()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveVirtHostExt

```
// HaveVirtHostExt()
// =================

boolean HaveVirtHostExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/extension/Havev8p4Debug

```
// Havev8p4Debug()
// ===============
// Returns TRUE if support for the Debugv8p4 feature is implemented and FALSE otherwise.

boolean Havev8p4Debug()
    return HasArchVersion(ARMv8p4);
```

Library pseudocode for shared/functions/extension/Havev8p8Debug

```
// Havev8p8Debug()
// ===============
// Returns TRUE if support for the Debugv8p8 feature is implemented and FALSE otherwise.

boolean Havev8p8Debug()
    return HasArchVersion(ARMv8p8);
```

Library pseudocode for shared/functions/extension/InsertIESBBeforeException

```
// InsertIESBBeforeException()
// ===========================
// Returns an implementation defined choice whether to insert an implicit error synchronization
// barrier before exception.
// If SCTLR_ELx.IESB is 1 when an exception is generated to ELx, any pending Unrecoverable
// SError interrupt must be taken before executing any instructions in the exception handler.
// However, this can be before the branch to the exception handler is made.

boolean InsertIESBBeforeException(bits(2) el)
    return (HaveIESB() &&
            boolean IMPLEMENTATION_DEFINED "Has Implicit Error Synchronization Barrier before Exception");
```

Library pseudocode for shared/functions/extension/IsG1ActivityMonitorImplemented

```
// Returns TRUE if a G1 activity monitor is implemented for the counter
// and FALSE otherwise.
boolean IsG1ActivityMonitorImplemented(integer i);
```

Library pseudocode for shared/functions/extension/IsG1ActivityMonitorOffsetImplemented

```
// Returns TRUE if a G1 activity monitor offset is implemented for the counter,
// and FALSE otherwise.
boolean IsG1ActivityMonitorOffsetImplemented(integer i);
```

Library pseudocode for shared/functions/externalaborts/HandleExternalAbort

```
// HandleExternalAbort()
// =====================
// Takes a Synchronous/Asynchronous abort based on fault.

HandleExternalAbort(PhysMemRetStatus memretstatus, boolean iswrite,
                    AddressDescriptor memaddrdesc, integer size,
                    AccessDescriptor accdesc)
    assert (memretstatus.statuscode IN {Fault_SyncExternal, Fault_AsyncExternal} ||
           (!HaveRASExt() && memretstatus.statuscode IN {Fault_SyncParity,
                                                         Fault_AsyncParity}));

    fault = NoFault();
    fault.statuscode = memretstatus.statuscode;
    fault.write = iswrite;
    fault.extflag = memretstatus.extflag;
    fault.acctype = memretstatus.acctype;
    // It is implementation specific whether External aborts signaled
    // in-band synchronously are taken synchronously or asynchronously
    if (IsExternalSyncAbort(fault) &&
            !IsExternalAbortTakenSynchronously(memretstatus, iswrite, memaddrdesc,
                                               size, accdesc)) then
        if fault.statuscode == Fault_SyncParity then
            fault.statuscode = Fault_AsyncParity;
        else
            fault.statuscode = Fault_AsyncExternal;

    if HaveRASExt() then
        fault.errortype = PEErrorState(memretstatus);
    else
        fault.errortype = bits(2) UNKNOWN;

    if IsExternalSyncAbort(fault) then
        if UsingAArch32() then
            AArch32.Abort(memaddrdesc.vaddress<31:0>, fault);
        else
            AArch64.Abort(memaddrdesc.vaddress, fault);

    else
        PendSErrorInterrupt(fault);
```

Library pseudocode for shared/functions/externalaborts/HandleExternalReadAbort

```
// HandleExternalReadAbort()
// =========================
// Wrapper function for HandleExternalAbort function in case of an External
// Abort on memory read.

HandleExternalReadAbort(PhysMemRetStatus memstatus, AddressDescriptor memaddrdesc,
                        integer size, AccessDescriptor accdesc)
    iswrite = FALSE;
    HandleExternalAbort(memstatus, iswrite, memaddrdesc, size, accdesc);
```

Library pseudocode for shared/functions/externalaborts/HandleExternalTTWAbort

```
// HandleExternalTTWAbort()
// ========================
// Take Asynchronous abort or update FaultRecord for Translation Table Walk
// based on PhysMemRetStatus.

FaultRecord HandleExternalTTWAbort(PhysMemRetStatus memretstatus, boolean iswrite,
                                   AddressDescriptor memaddrdesc,
                                   AccessDescriptor accdesc, integer size,
                                   FaultRecord input_fault)
    output_fault = input_fault;
    output_fault.extflag = memretstatus.extflag;
    output_fault.statuscode = memretstatus.statuscode;
    if (IsExternalSyncAbort(output_fault) &&
            !IsExternalAbortTakenSynchronously(memretstatus, iswrite,
                                               memaddrdesc,
                                               size, accdesc)) then
        if output_fault.statuscode == Fault_SyncParity then
            output_fault.statuscode = Fault_AsyncParity;
        else
            output_fault.statuscode = Fault_AsyncExternal;

    // If a synchronous fault is on a translation table walk, then update
    // the fault type
    if IsExternalSyncAbort(output_fault) then
        if output_fault.statuscode == Fault_SyncParity then
            output_fault.statuscode = Fault_SyncParityOnWalk;
        else
            output_fault.statuscode = Fault_SyncExternalOnWalk;
    if HaveRASExt() then
        output_fault.errortype = PEErrorState(memretstatus);
    else
        output_fault.errortype = bits(2) UNKNOWN;
    if !IsExternalSyncAbort(output_fault) then
        PendSErrorInterrupt(output_fault);
        output_fault.statuscode = Fault_None;
    return output_fault;
```

Library pseudocode for shared/functions/externalaborts/HandleExternalWriteAbort

```
// HandleExternalWriteAbort()
// ==========================
// Wrapper function for HandleExternalAbort function in case of an External
// Abort on memory write.

HandleExternalWriteAbort(PhysMemRetStatus memstatus, AddressDescriptor memaddrdesc,
                         integer size, AccessDescriptor accdesc)
    iswrite = TRUE;
    HandleExternalAbort(memstatus, iswrite, memaddrdesc, size, accdesc);
```

Library pseudocode for shared/functions/externalaborts/IsExternalAbortTakenSynchronously

```
// Return an implementation specific value:
// TRUE if the fault returned for the access can be taken synchronously,
// FALSE otherwise.
//
// This might vary between accesses, for example depending on the error type
// or memory type being accessed.
// External aborts on data accesses and translation table walks on data accesses
// can be either synchronous or asynchronous.
//
// When FEAT_DoubleFault is not implemented, External aborts on instruction
// fetches and translation table walks on instruction fetches can be either
// synchronous or asynchronous.
// When FEAT_DoubleFault is implemented, all External abort exceptions on
// instruction fetches and translation table walks on instruction fetches
// must be synchronous.
boolean IsExternalAbortTakenSynchronously(PhysMemRetStatus memstatus,
                                          boolean iswrite,
                                          AddressDescriptor desc,
                                          integer size,
                                          AccessDescriptor accdesc);
```

Library pseudocode for shared/functions/externalaborts/PEErrorState

```
constant bits(2) Sync_UC   = '10'; // Synchronous Uncontainable
constant bits(2) Sync_UER  = '00'; // Synchronous Recoverable
constant bits(2) Sync_UEO  = '11'; // Synchronous Restartable
constant bits(2) ASync_UC  = '00'; // ASynchronous Uncontainable
constant bits(2) ASync_UEU = '01'; // ASynchronous Unrecoverable
constant bits(2) ASync_UER = '11'; // ASynchronous Recoverable
constant bits(2) ASync_UEO = '10'; // ASynchronous Restartable

bits(2) PEErrorState(PhysMemRetStatus memstatus);
```

Library pseudocode for shared/functions/externalaborts/PendSErrorInterrupt

```
// Pend the SError.
PendSErrorInterrupt(FaultRecord fault);
```

Library pseudocode for shared/functions/float/bfloat/BFDotAdd

```
// BFDotAdd()
// ==========
// BFloat16 2-way dot-product and add to single-precision
// result = addend + op1_a*op2_a + op1_b*op2_b

bits(32) BFDotAdd(bits(32) addend, bits(16) op1_a, bits(16) op1_b,
                  bits(16) op2_a, bits(16) op2_b, FPCRType fpcr_in)
    FPCRType fpcr = fpcr_in;

    bits(32) prod;

    bits(32) result;
    if !HaveEBF16() || fpcr.EBF == '0' then   // Standard BFloat16 behaviors
        prod = FPAdd_BF16(BFMulH(op1_a, op2_a), BFMulH(op1_b, op2_b));
        result = FPAdd_BF16(addend, prod);
    else                                      // Extended BFloat16 behaviors
        boolean isbfloat16 = TRUE;
        boolean fpexc = FALSE;  // Do not generate floating-point exceptions
        fpcr.DN = '1';          // Generate default NaN values
        prod = FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc);
        result = FPAdd(addend, prod, fpcr, fpexc);

    return result;
```

Library pseudocode for shared/functions/float/bfloat/BFMatMulAdd

```
// BFMatMulAdd()
// =============
// BFloat16 matrix multiply and add to single-precision matrix
// result[2, 2] = addend[2, 2] + (op1[2, 4] * op2[4, 2])

bits(N) BFMatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2)

    assert N == 128;

    bits(N) result;
    bits(32) sum;

    for i = 0 to 1
        for j = 0 to 1
            sum = Elem[addend, 2*i + j, 32];
            for k = 0 to 1
                bits(16) elt1_a = Elem[op1, 4*i + 2*k + 0, 16];
                bits(16) elt1_b = Elem[op1, 4*i + 2*k + 1, 16];
                bits(16) elt2_a = Elem[op2, 4*j + 2*k + 0, 16];
                bits(16) elt2_b = Elem[op2, 4*j + 2*k + 1, 16];
                sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, FPCR[]);
            Elem[result, 2*i + j, 32] = sum;

    return result;
```

Library pseudocode for shared/functions/float/bfloat/BFMulAddH

```
// BFMulAddH()
// ===========
// Used by BFMLALB and BFMLALT instructions.

bits(N) BFMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2, FPCRType fpcr_in)
    bits(N) value1 = op1 : Zeros(N DIV 2);
    bits(N) value2 = op2 : Zeros(N DIV 2);
    FPCRType fpcr = fpcr_in;
    boolean altfp = HaveAltFP() && fpcr.AH == '1';  // When TRUE:
    boolean fpexc = !altfp;                         //     Do not generate floating point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11';             //     Flush denormal input and output to zero
    if altfp then fpcr.RMode    = '00';             //     Use RNE rounding mode
    return FPMulAdd(addend, value1, value2, fpcr, fpexc);
```

Library pseudocode for shared/functions/float/bfloat/BFMulH

```
// BFMulH()
// ========
// BFloat16 widening multiply to single-precision following BFloat16
// computation behaviors.

bits(32) BFMulH(bits(16) op1, bits(16) op2)

    bits(32) result;

    FPCRType fpcr = FPCR[];
    (type1,sign1,value1) = BFUnpack(op1);
    (type2,sign2,value2) = BFUnpack(op2);
    if type1 == FPType_QNaN || type2 == FPType_QNaN then
        result = FPDefaultNaN(fpcr, 32);
    else
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN(fpcr, 32);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, 32);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2, 32);
        else
            result = BFRound(value1*value2);

    return result;
```

Library pseudocode for shared/functions/float/bfloat/BFNeg

```
// BFNeg()
// =======

bits(16) BFNeg(bits(16) op)
    return NOT(op<15>) : op<14:0>;
```

Library pseudocode for shared/functions/float/bfloat/BFRound

```
// BFRound()
// =========
// Converts a real number OP into a single-precision value using the
// Round to Odd rounding mode and following BFloat16 computation behaviors.

bits(32) BFRound(real op)

    assert op != 0.0;
    bits(32) result;

    // Format parameters - minimum exponent, numbers of exponent and fraction bits.
    minimum_exp = -126;  E = 8;  F = 23;

    // Split value into sign, unrounded mantissa and exponent.
    bit sign;
    real mantissa;
    if op < 0.0 then
        sign = '1';  mantissa = -op;
    else
        sign = '0';  mantissa = op;
    exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0;  exponent = exponent - 1;
    while mantissa >= 2.0 do
        mantissa = mantissa / 2.0;  exponent = exponent + 1;

    // Fixed Flush-to-zero.
    if exponent < minimum_exp then
        return FPZero(sign, 32);

    // Start creating the exponent value for the result. Start by biasing the actual exponent
    // so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
    biased_exp = Max((exponent - minimum_exp) + 1, 0);
    if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

    // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
    int_mant = RoundDown(mantissa * 2.0^F);  // < 2.0^F if biased_exp == 0, >= 2.0^F if not
    error = mantissa * 2.0^F - Real(int_mant);

    // Round to Odd
    if error != 0.0 then
        int_mant<0> = '1';

    // Deal with overflow and generate result.
    if biased_exp >= 2^E - 1 then
        result = FPInfinity(sign, 32);      // Overflows generate appropriately-signed Infinity
    else
        result = sign : biased_exp<30-F:0> : int_mant<F-1:0>;

    return result;
```

Library pseudocode for shared/functions/float/bfloat/BFUnpack

```
// BFUnpack()
// ==========
// Unpacks a BFloat16 or single-precision value into its type,
// sign bit and real number that it represents.
// The real number result has the correct sign for numbers and infinities,
// is very large in magnitude for infinities, and is 0.0 for NaNs.
// (These values are chosen to simplify the description of
// comparisons and conversions.)

(FPType, bit, real) BFUnpack(bits(N) fpval)

    assert N IN {16,32};

    bit sign;
    bits(8) exp;
    bits(23) frac;
    if N == 16 then
        sign   = fpval<15>;
        exp    = fpval<14:7>;
        frac   = fpval<6:0> : Zeros(16);
    else  // N == 32
        sign   = fpval<31>;
        exp    = fpval<30:23>;
        frac   = fpval<22:0>;

    FPType fptype;
    real value;
    if IsZero(exp) then
        fptype = FPType_Zero;  value = 0.0;    // Fixed Flush to Zero
    elsif IsOnes(exp) then
        if IsZero(frac) then
            fptype = FPType_Infinity;  value = 2.0^1000000;
        else    // no SNaN for BF16 arithmetic
            fptype = FPType_QNaN; value = 0.0;
    else
        fptype = FPType_Nonzero;
        value = 2.0^(UInt(exp)-127) * (1.0 + Real(UInt(frac)) * 2.0^-23);

    if sign == '1' then value = -value;

    return (fptype, sign, value);
```

Library pseudocode for shared/functions/float/bfloat/FPAdd_BF16

```
// FPAdd_BF16()
// ============
// Single-precision add following BFloat16 computation behaviors.

bits(32) FPAdd_BF16(bits(32) op1, bits(32) op2)

    bits(32) result;

    FPCRType fpcr = FPCR[];
    (type1,sign1,value1) = BFUnpack(op1);
    (type2,sign2,value2) = BFUnpack(op2);
    if type1 == FPType_QNaN || type2 == FPType_QNaN then
        result = FPDefaultNaN(fpcr, 32);
    else
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN(fpcr, 32);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0', 32);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1', 32);
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1, 32);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then
                result = FPZero('0', 32);    // Positive sign when Round to Odd
            else
                result = BFRound(result_value);

    return result;
```

Library pseudocode for shared/functions/float/bfloat/FPConvertBF

```
// FPConvertBF()
// =============
// Converts a single-precision OP to BFloat16 value with using rounding mode of
// Round to Nearest Even when executed from AArch64 state and
// FPCR.AH == '1', otherwise rounding is controlled by FPCR/FPSCR.

bits(16) FPConvertBF(bits(32) op, FPCRType fpcr_in, FPRounding rounding_in)

    FPCRType fpcr = fpcr_in;
    FPRounding rounding = rounding_in;
    bits(32) result;                                // BF16 value in top 16 bits
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    boolean fpexc = !altfp;                         // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11';             // Flush denormal input and output to zero
    if altfp then rounding = FPRounding_TIEEVEN;    // Use RNE rounding mode

    // Unpack floating-point operand, with always flush-to-zero if fpcr.AH == '1'.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        if fpcr.DN == '1' then
            result = FPDefaultNaN(fpcr, 32);
        else
            result = FPConvertNaN(op, 32);
        if fptype == FPType_SNaN then
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
    elsif fptype == FPType_Infinity then
        result = FPInfinity(sign, 32);
    elsif fptype == FPType_Zero then
        result = FPZero(sign, 32);
    else
        result = FPRoundBF(value, fpcr, rounding, fpexc);

    // Returns correctly rounded BF16 value from top 16 bits
    return result<31:16>;

// FPConvertBF()
// =============
// Converts a single-precision operand to BFloat16 value.

bits(16) FPConvertBF(bits(32) op, FPCRType fpcr)
    return FPConvertBF(op, fpcr, FPRoundingMode(fpcr));
```

Library pseudocode for shared/functions/float/bfloat/FPRoundBF

```
// FPRoundBF()
// ===========
// Converts a real number OP into a BFloat16 value using the supplied
// rounding mode RMODE. The 'fpexc' argument controls the generation of
// floating-point exceptions.

bits(32) FPRoundBF(real op, FPCRType fpcr, FPRounding rounding, boolean fpexc)
    boolean isbfloat16 = TRUE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc, 32);
```

Library pseudocode for shared/functions/float/fixedtofp/FixedToFP

```
// FixedToFP()
// ===========

// Convert M-bit fixed point 'op' with FBITS fractional bits to
// N-bit precision floating point, controlled by UNSIGNED and ROUNDING.

bits(N) FixedToFP(bits(M) op, integer fbits, boolean unsigned, FPCRType fpcr, FPRounding rounding, intege

    assert N IN {16,32,64};
    assert M IN {16,32,64};
    bits(N) result;
    assert fbits >= 0;
    assert rounding != FPRounding_ODD;

    // Correct signed-ness
    int_operand = Int(op, unsigned);

    // Scale by fractional bits and generate a real value
    real_operand = Real(int_operand) / 2.0^fbits;

    if real_operand == 0.0 then
        result = FPZero('0', N);
    else
        result = FPRound(real_operand, fpcr, rounding, N);

    return result;
```

Library pseudocode for shared/functions/float/fpabs/FPAbs

```
// FPAbs()
// =======

bits(N) FPAbs(bits(N) op)

    assert N IN {16,32,64};
    if !UsingAArch32() && HaveAltFP() then
        FPCRType fpcr = FPCR[];
        if fpcr.AH == '1' then
            (fptype, -, -) = FPUnpack(op, fpcr, FALSE);
            if fptype IN {FPType_SNaN, FPType_QNaN} then
                return op;        // When fpcr.AH=1, sign of NaN has no consequence

    return '0' : op<N-2:0>;
```

Library pseudocode for shared/functions/float/fpadd/FPAdd

```
// FPAdd()
// =======

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCRType fpcr)
    boolean fpexc = TRUE;       // Generate floating-point exceptions
    return FPAdd(op1, op2, fpcr, fpexc);

// FPAdd()
// =======

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCRType fpcr, boolean fpexc)

    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);

    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);
    if !done then
        inf1  = (type1 == FPType_Infinity);  inf2  = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);      zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN(fpcr, N);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0', N);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1', N);
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1, N);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);

        if fpexc then FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpcompare/FPCompare

```
// FPCompare()
// ===========

bits(4) FPCompare(bits(N) op1, bits(N) op2, boolean signal_nans, FPCRType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    bits(4) result;
    if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
        result = '0011';
        if type1 == FPType_SNaN || type2 == FPType_SNaN || signal_nans then
            FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        if value1 == value2 then
            result = '0110';
        elsif value1 < value2 then
            result = '1000';
        else  // value1 > value2
            result = '0010';

        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpcompareeq/FPCompareEQ

```
// FPCompareEQ()
// =============

boolean FPCompareEQ(bits(N) op1, bits(N) op2, FPCRType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    boolean result;
    if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
        result = FALSE;
        if type1 == FPType_SNaN || type2 == FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 == value2);
        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpcomparege/FPCompareGE

```
// FPCompareGE()
// =============

boolean FPCompareGE(bits(N) op1, bits(N) op2, FPCRType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    boolean result;
    if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 >= value2);
        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpcomparegt/FPCompareGT

```
// FPCompareGT()
// =============

boolean FPCompareGT(bits(N) op1, bits(N) op2, FPCRType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    boolean result;
    if type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN} then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 > value2);
        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpconvert/FPConvert

```
// FPConvert()
// ===========

// Convert floating point 'op' with N-bit precision to M-bit precision,
// with rounding controlled by ROUNDING.
// This is used by the FP-to-FP conversion instructions and so for
// half-precision data ignores FZ16, but observes AHP.

bits(M) FPConvert(bits(N) op, FPCRType fpcr, FPRounding rounding, integer M)

    assert M IN {16,32,64};
    assert N IN {16,32,64};
    bits(M) result;

    // Unpack floating-point operand optionally with flush-to-zero.
    (fptype,sign,value) = FPUnpackCV(op, fpcr);

    alt_hp = (M == 16) && (fpcr.AHP == '1');

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        if alt_hp then
            result = FPZero(sign, M);
        elsif fpcr.DN == '1' then
            result = FPDefaultNaN(fpcr, M);
        else
            result = FPConvertNaN(op, M);
        if fptype == FPType_SNaN || alt_hp then
            FPProcessException(FPExc_InvalidOp,fpcr);
    elsif fptype == FPType_Infinity then
        if alt_hp then
            result = sign:Ones(M-1);
            FPProcessException(FPExc_InvalidOp, fpcr);
        else
            result = FPInfinity(sign, M);
    elsif fptype == FPType_Zero then
        result = FPZero(sign, M);
    else
        result = FPRoundCV(value, fpcr, rounding, M);
        FPProcessDenorm(fptype, N, fpcr);

    return result;

// FPConvert()
// ===========

bits(M) FPConvert(bits(N) op, FPCRType fpcr, integer M)
    return FPConvert(op, fpcr, FPRoundingMode(fpcr), M);
```

Library pseudocode for shared/functions/float/fpconvertnan/FPConvertNaN

```
// FPConvertNaN()
// ==============
// Converts a NaN of one floating-point type to another

bits(M) FPConvertNaN(bits(N) op, integer M)

    assert N IN {16,32,64};
    assert M IN {16,32,64};
    bits(M) result;
    bits(51) frac;

    sign = op<N-1>;

    // Unpack payload from input NaN
    case N of
        when 64 frac = op<50:0>;
        when 32 frac = op<21:0>:Zeros(29);
        when 16 frac = op<8:0>:Zeros(42);

    // Repack payload into output NaN, while
    // converting an SNaN to a QNaN.
    case M of
        when 64 result = sign:Ones(M-52):frac;
        when 32 result = sign:Ones(M-23):frac<50:29>;
        when 16 result = sign:Ones(M-10):frac<50:42>;

    return result;
```

Library pseudocode for shared/functions/float/fpcrtype/FPCRType

```
type FPCRType;
```

Library pseudocode for shared/functions/float/fpdecoderm/FPDecodeRM

```
// FPDecodeRM()
// ============

// Decode most common AArch32 floating-point rounding encoding.

FPRounding FPDecodeRM(bits(2) rm)

    FPRounding result;
    case rm of
        when '00' result = FPRounding_TIEAWAY; // A
        when '01' result = FPRounding_TIEEVEN; // N
        when '10' result = FPRounding_POSINF;  // P
        when '11' result = FPRounding_NEGINF;  // M

    return result;
```

Library pseudocode for shared/functions/float/fpdecoderounding/FPDecodeRounding

```
// FPDecodeRounding()
// ==================

// Decode floating-point rounding mode and common AArch64 encoding.

FPRounding FPDecodeRounding(bits(2) rmode)
    case rmode of
        when '00' return FPRounding_TIEEVEN; // N
        when '01' return FPRounding_POSINF;  // P
        when '10' return FPRounding_NEGINF;  // M
        when '11' return FPRounding_ZERO;    // Z
```

Library pseudocode for shared/functions/float/fpdefaultnan/FPDefaultNaN

```
// FPDefaultNaN()
// ==============

bits(N) FPDefaultNaN(integer N)
    FPCRType fpcr = FPCR[];
    return FPDefaultNaN(fpcr, N);

bits(N) FPDefaultNaN(FPCRType fpcr, integer N)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    bit sign = if HaveAltFP() && !UsingAArch32() then fpcr.AH else '0';

    bits(E) exp  = Ones(E);
    bits(F) frac = '1':Zeros(F-1);

    return sign : exp : frac;
```

Library pseudocode for shared/functions/float/fpdiv/FPDiv

```
// FPDiv()
// =======

bits(N) FPDiv(bits(N) op1, bits(N) op2, FPCRType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);

    if !done then
        inf1  = type1 == FPType_Infinity;
        inf2  = type2 == FPType_Infinity;
        zero1 = type1 == FPType_Zero;
        zero2 = type2 == FPType_Zero;

        if (inf1 && inf2) || (zero1 && zero2) then
            result = FPDefaultNaN(fpcr, N);
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif inf1 || zero2 then
            result = FPInfinity(sign1 EOR sign2, N);
            if !inf1 then FPProcessException(FPExc_DivideByZero, fpcr);
        elsif zero1 || inf2 then
            result = FPZero(sign1 EOR sign2, N);
        else
            result = FPRound(value1/value2, fpcr, N);

        if !zero2 then
            FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpdot/FPDot

```
// FPDot()
// =======
// Calculates single-precision result of 2-way 16-bit floating-point dot-product
// with a single rounding.
// The 'fpcr' argument supplies the FPCR control bits and 'isbfloat16'
// determines whether input operands are BFloat16 or half-precision type.
// and 'fpexc' controls the generation of floating-point exceptions.

bits(N) FPDot(bits(N DIV 2) op1_a, bits(N DIV 2) op1_b, bits(N DIV 2) op2_a,
              bits(N DIV 2) op2_b, FPCRType fpcr, boolean isbfloat16)
    boolean fpexc = TRUE;       // Generate floating-point exceptions
    return FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc);

bits(N) FPDot(bits(N DIV 2) op1_a, bits(N DIV 2) op1_b, bits(N DIV 2) op2_a,
              bits(N DIV 2) op2_b, FPCRType fpcr_in, boolean isbfloat16, boolean fpexc)
    FPCRType fpcr = fpcr_in;

    assert N == 32;
    bits(N) result;
    boolean done;
    fpcr.AHP = '0';           // Ignore alternative half-precision option
    rounding = FPRoundingMode(fpcr);

    (type1_a,sign1_a,value1_a) = FPUnpackBase(op1_a, fpcr, fpexc, isbfloat16);
    (type1_b,sign1_b,value1_b) = FPUnpackBase(op1_b, fpcr, fpexc, isbfloat16);
    (type2_a,sign2_a,value2_a) = FPUnpackBase(op2_a, fpcr, fpexc, isbfloat16);
    (type2_b,sign2_b,value2_b) = FPUnpackBase(op2_b, fpcr, fpexc, isbfloat16);

    inf1_a = (type1_a == FPType_Infinity); zero1_a = (type1_a == FPType_Zero);
    inf1_b = (type1_b == FPType_Infinity); zero1_b = (type1_b == FPType_Zero);
    inf2_a = (type2_a == FPType_Infinity); zero2_a = (type2_a == FPType_Zero);
    inf2_b = (type2_b == FPType_Infinity); zero2_b = (type2_b == FPType_Zero);

    (done,result) = FPProcessNaNs4(type1_a, type1_b, type2_a, type2_b,
                                   op1_a, op1_b, op2_a, op2_b, fpcr, fpexc);

    if (((inf1_a && zero2_a) || (zero1_a && inf2_a)) &&
        ((inf1_b && zero2_b) || (zero1_b && inf2_b))) then
        result = FPDefaultNaN(fpcr, N);
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);

    if !done then
        // Determine sign and type products will have if it does not cause an Invalid
        // Operation.
        signPa = sign1_a EOR sign2_a;
        signPb = sign1_b EOR sign2_b;
        infPa = inf1_a || inf2_a;
        infPb = inf1_b || inf2_b;
        zeroPa = zero1_a || zero2_a;
        zeroPb = zero1_b || zero2_b;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero
        // by infinity and additions of opposite-signed infinities.
        invalidop = ((inf1_a && zero2_a) || (zero1_a && inf2_a) ||
            (inf1_b && zero2_b) || (zero1_b && inf2_b) || (infPa && infPb && signPa != signPb));

        if invalidop then
            result = FPDefaultNaN(fpcr, N);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);

       // Other cases involving infinities produce an infinity of the same sign.
        elsif (infPa && signPa == '0') || (infPb && signPb == '0') then
            result = FPInfinity('0', N);
        elsif (infPa && signPa == '1') || (infPb && signPb == '1') then
            result = FPInfinity('1', N);

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroPa && zeroPb && signPa == signPb then
            result = FPZero(signPa, N);
```

```
        // Otherwise calculate fused sum of products and round it.
        else
            result_value = (value1_a * value2_a) + (value1_b * value2_b);
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);

    return result;
```

Library pseudocode for shared/functions/float/fpdot/FPDotAdd

```
// FPDotAdd()
// ==========
// Half-precision 2-way dot-product and add to single-precision.

bits(N) FPDotAdd(bits(N) addend, bits(N DIV 2) op1_a, bits(N DIV 2) op1_b,
                 bits(N DIV 2) op2_a, bits(N DIV 2) op2_b, FPCRType fpcr)
    assert N == 32;

    bits(N) prod;
    boolean isbfloat16 = FALSE;
    boolean fpexc = TRUE;     // Generate floating-point exceptions
    prod = FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc);
    result = FPAdd(addend, prod, fpcr, fpexc);

    return result;
```

Library pseudocode for shared/functions/float/fpdot/FPDotAdd_ZA

```
// FPDotAdd_ZA()
// =============
// Half-precision 2-way dot-product and add to single-precision
// for SME ZA-targeting instructions.

bits(N) FPDotAdd_ZA(bits(N) addend, bits(N DIV 2) op1_a, bits(N DIV 2) op1_b,
                    bits(N DIV 2) op2_a, bits(N DIV 2) op2_b, FPCRType fpcr_in)
    FPCRType fpcr = fpcr_in;
    assert N == 32;

    bits(N) prod;
    boolean isbfloat16 = FALSE;
    boolean fpexc = FALSE;    // Do not generate floating-point exceptions
    fpcr.DN = '1';            // Generate default NaN values
    prod = FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc);
    result = FPAdd(addend, prod, fpcr, fpexc);

    return result;
```

Library pseudocode for shared/functions/float/fpexc/FPExc

```
enumeration FPExc       {FPExc_InvalidOp, FPExc_DivideByZero, FPExc_Overflow,
                         FPExc_Underflow, FPExc_Inexact, FPExc_InputDenorm};
```

Library pseudocode for shared/functions/float/fpinfinity/FPInfinity

```
// FPInfinity()
// ============

bits(N) FPInfinity(bit sign, integer N)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    bits(E) exp  = Ones(E);
    bits(F) frac = Zeros(F);

    return sign : exp : frac;
```

Library pseudocode for shared/functions/float/fpmatmul/FPMatMulAdd

```
// FPMatMulAdd()
// =============
//
// Floating point matrix multiply and add to same precision matrix
// result[2, 2] = addend[2, 2] + (op1[2, 2] * op2[2, 2])

bits(N) FPMatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, integer esize, FPCRType fpcr)

    assert N == esize * 2 * 2;
    bits(N)  result;
    bits(esize) prod0, prod1, sum;

    for i = 0 to 1
        for j = 0 to 1
            sum   = Elem[addend, 2*i + j, esize];
            prod0 = FPMul(Elem[op1, 2*i + 0, esize],
                          Elem[op2, 2*j + 0, esize], fpcr);
            prod1 = FPMul(Elem[op1, 2*i + 1, esize],
                          Elem[op2, 2*j + 1, esize], fpcr);
            sum   = FPAdd(sum, FPAdd(prod0, prod1, fpcr), fpcr);
            Elem[result, 2*i + j, esize] = sum;

    return result;
```

Library pseudocode for shared/functions/float/fpmax/FPMax

```
// FPMax()
// =======

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCRType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    return FPMax(op1, op2, fpcr, altfp);

// FPMax()
// =======
// Compare two inputs and return the larger value after rounding. The
// 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative floating-point behaviour.

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCRType fpcr_in, boolean altfp)

    assert N IN {16,32,64};
    boolean done;
    bits(N) result;
    FPCRType fpcr = fpcr_in;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    if altfp && type1 == FPType_Zero && type2 == FPType_Zero && sign1 != sign2 then
        // Alternate handling of zeros with differing sign
        return FPZero(sign2, N);

    if altfp && (type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN}) then
        // Alternate handling of NaN inputs
        FPProcessException(FPExc_InvalidOp, fpcr);
        result = if type2 == FPType_Zero then FPZero(sign2, N) else op2;
        done = TRUE;
    else
        (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);

    if !done then
        FPType fptype;
        bit sign;
        real value;
        if value1 > value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPType_Infinity then
            result = FPInfinity(sign, N);
        elsif fptype == FPType_Zero then
            sign = sign1 AND sign2;         // Use most positive sign
            result = FPZero(sign, N);
        else
            // The use of FPRound() covers the case where there is a trapped underflow exception
            // for a denormalized number even though the result is exact.
            rounding = FPRoundingMode(fpcr);
            if altfp then    // Denormal output is not flushed to zero
                fpcr.FZ = '0';
                fpcr.FZ16 = '0';

            result = FPRound(value, fpcr, rounding, TRUE, N);

        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpmaxnormal/FPMaxNormal

```
// FPMaxNormal()
// =============

bits(N) FPMaxNormal(bit sign, integer N)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp  = Ones(E-1):'0';
    frac = Ones(F);

    return sign : exp : frac;
```

Library pseudocode for shared/functions/float/fpmaxnum/FPMaxNum

```
// FPMaxNum()
// ==========

bits(N) FPMaxNum(bits(N) op1_in, bits(N) op2_in, FPCRType fpcr)

    assert N IN {16,32,64};
    bits(N) op1 = op1_in;
    bits(N) op2 = op2_in;
    (type1,-,-) = FPUnpack(op1, fpcr);
    (type2,-,-) = FPUnpack(op2, fpcr);

    boolean type1_nan = type1 IN {FPType_QNaN, FPType_SNaN};
    boolean type2_nan = type2 IN {FPType_QNaN, FPType_SNaN};
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';

    if !(altfp && type1_nan && type2_nan) then
        // Treat a single quiet-NaN as -Infinity.
        if type1 == FPType_QNaN && type2 != FPType_QNaN then
            op1 = FPInfinity('1', N);
        elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
            op2 = FPInfinity('1', N);

    altfmaxfmin = FALSE;    // Restrict use of FMAX/FMIN NaN propagation rules
    result = FPMax(op1, op2, fpcr, altfmaxfmin);

    return result;
```

Library pseudocode for shared/functions/float/fpmerge/IsMerging

```
// IsMerging()
// ===========
// Returns TRUE if the output elements other than the lowest are taken from
// the destination register.

boolean IsMerging(FPCRType fpcr)
    bit nep = if HaveSME() && PSTATE.SM == '1' && !IsFullA64Enabled() then '0' else fpcr.NEP;
    return HaveAltFP() && !UsingAArch32() && nep == '1';
```

Library pseudocode for shared/functions/float/fpmin/FPMin

```
// FPMin()
// =======

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCRType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    return FPMin(op1, op2, fpcr, altfp);

// FPMin()
// =======
// Compare two operands and return the smaller operand after rounding. The
// 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative behaviour.

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCRType fpcr_in, boolean altfp)

    assert N IN {16,32,64};
    boolean done;
    bits(N) result;
    FPCRType fpcr = fpcr_in;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if altfp && type1 == FPType_Zero && type2 == FPType_Zero && sign1 != sign2 then
        // Alternate handling of zeros with differing sign
        return FPZero(sign2, N);

    if altfp && (type1 IN {FPType_SNaN, FPType_QNaN} || type2 IN {FPType_SNaN, FPType_QNaN}) then
        // Alternate handling of NaN inputs
        FPProcessException(FPExc_InvalidOp, fpcr);
        result = if type2 == FPType_Zero then FPZero(sign2, N) else op2;
        done = TRUE;
    else
        (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);

    if !done then
        FPType fptype;
        bit sign;
        real value;
        FPRounding rounding;
        if value1 < value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPType_Infinity then
            result = FPInfinity(sign, N);
        elsif fptype == FPType_Zero then
            sign = sign1 OR sign2;              // Use most negative sign
            result = FPZero(sign, N);
        else
            // The use of FPRound() covers the case where there is a trapped underflow exception
            // for a denormalized number even though the result is exact.
            rounding = FPRoundingMode(fpcr);
            if altfp then    // Denormal output is not flushed to zero
                fpcr.FZ = '0';
                fpcr.FZ16 = '0';

            result = FPRound(value, fpcr, rounding, TRUE, N);

        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpminnum/FPMinNum

```
// FPMinNum()
// ==========

bits(N) FPMinNum(bits(N) op1_in, bits(N) op2_in, FPCRType fpcr)

    assert N IN {16,32,64};
    bits(N) op1 = op1_in;
    bits(N) op2 = op2_in;
    (type1,-,-) = FPUnpack(op1, fpcr);
    (type2,-,-) = FPUnpack(op2, fpcr);

    boolean type1_nan = type1 IN {FPType_QNaN, FPType_SNaN};
    boolean type2_nan = type2 IN {FPType_QNaN, FPType_SNaN};
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';

    if !(altfp && type1_nan && type2_nan) then
        // Treat a single quiet-NaN as +Infinity.
        if type1 == FPType_QNaN && type2 != FPType_QNaN then
            op1 = FPInfinity('0', N);
        elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
            op2 = FPInfinity('0', N);

    altfmaxfmin = FALSE;    // Restrict use of FMAX/FMIN NaN propagation rules
    result = FPMin(op1, op2, fpcr, altfmaxfmin);

    return result;
```

Library pseudocode for shared/functions/float/fpmul/FPMul

```
// FPMul()
// =======

bits(N) FPMul(bits(N) op1, bits(N) op2, FPCRType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN(fpcr, N);
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, N);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2, N);
        else
            result = FPRound(value1*value2, fpcr, N);

        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpmuladd/FPMulAdd

```
// FPMulAdd()
// ==========

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr)
    boolean fpexc = TRUE;       // Generate floating-point exceptions
    return FPMulAdd(addend, op1, op2, fpcr, fpexc);

// FPMulAdd()
// ==========
//
// Calculates addend + op1*op2 with a single rounding. The 'fpcr' argument
// supplies the FPCR control bits, and 'fpexc' controls the generation of
// floating-point exceptions.

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2,
                 FPCRType fpcr, boolean fpexc)

    assert N IN {16,32,64};

    (typeA,signA,valueA) = FPUnpack(addend, fpcr, fpexc);
    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
    rounding = FPRoundingMode(fpcr);
    inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
    inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);

    (done,result) = FPProcessNaNs3(typeA, type1, type2, addend, op1, op2, fpcr, fpexc);

    if !(HaveAltFP() && !UsingAArch32() && fpcr.AH == '1') then
        if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
            result = FPDefaultNaN(fpcr, N);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);

    if !done then
        infA = (typeA == FPType_Infinity); zeroA = (typeA == FPType_Zero);

        // Determine sign and type product will have if it does not cause an
        // Invalid Operation.
        signP = sign1 EOR sign2;
        infP  = inf1 || inf2;
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero
        // by infinity and additions of opposite-signed infinities.
        invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);

        if invalidop then
            result = FPDefaultNaN(fpcr, N);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0', N);
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1', N);

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero(signA, N);

        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);

        if !invalidop && fpexc then
```

```
            FPProcessDenorms3(typeA, type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpmuladd/FPMulAdd_ZA

```
// FPMulAdd_ZA()
// =============
// Calculates addend + op1*op2 with a single rounding for SME ZA-targeting
// instructions.

bits(N) FPMulAdd_ZA(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr_in)
    FPCRType fpcr = fpcr_in;
    boolean fpexc = FALSE;      // Do not generate floating-point exceptions
    fpcr.DN = '1';              // Generate default NaN values
    return FPMulAdd(addend, op1, op2, fpcr, fpexc);
```

Library pseudocode for shared/functions/float/fpmuladdh/FPMulAddH

```
// FPMulAddH()
// ===========
// Calculates addend + op1*op2.

bits(N) FPMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2, FPCRType fpcr)
    boolean fpexc = TRUE;       // Generate floating-point exceptions
    return FPMulAddH(addend, op1, op2, fpcr, fpexc);

// FPMulAddH()
// ===========
// Calculates addend + op1*op2.

bits(N) FPMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2,
                  FPCRType fpcr, boolean fpexc)

    assert N == 32;
    rounding = FPRoundingMode(fpcr);
    (typeA,signA,valueA) = FPUnpack(addend, fpcr, fpexc);
    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
    inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
    inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);

    (done,result) = FPProcessNaNs3H(typeA, type1, type2, addend, op1, op2, fpcr, fpexc);

    if !(HaveAltFP() && !UsingAArch32() && fpcr.AH == '1') then
        if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
            result = FPDefaultNaN(fpcr, N);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);

    if !done then
        infA = (typeA == FPType_Infinity); zeroA = (typeA == FPType_Zero);

        // Determine sign and type product will have if it does not cause an
        // Invalid Operation.
        signP = sign1 EOR sign2;
        infP = inf1 || inf2;
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
        // additions of opposite-signed infinities.
        invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);

        if invalidop then
            result = FPDefaultNaN(fpcr, N);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);

        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0', N);
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1', N);

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero(signA, N);

        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);

        if !invalidop && fpexc then
            FPProcessDenorm(typeA, N, fpcr);
```

```
    return result;
```

Library pseudocode for shared/functions/float/fpmuladdh/FPProcessNaNs3H

```
// FPProcessNaNs3H()
// =================

(boolean, bits(N)) FPProcessNaNs3H(FPType type1, FPType type2, FPType type3,
                                   bits(N) op1, bits(N DIV 2) op2, bits(N DIV 2) op3,
                                   FPCRType fpcr, boolean fpexc)

    assert N IN {32,64};

    bits(N) result;
    FPType type_nan;
    // When TRUE, use alternative NaN propagation rules.
    boolean altfp   = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    boolean op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};
    boolean op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};
    boolean op3_nan = type3 IN {FPType_SNaN, FPType_QNaN};
    if altfp then
        if (type1 == FPType_SNaN || type2 == FPType_SNaN || type3 == FPType_SNaN) then
            type_nan = FPType_SNaN;
        else
            type_nan = FPType_QNaN;

    boolean done;
    if altfp && op1_nan && op2_nan && op3_nan then          // <n> register NaN selected
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type_nan, op2, fpcr, fpexc), N);
    elsif altfp && op2_nan && (op1_nan || op3_nan) then     // <n> register NaN selected
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type_nan, op2, fpcr, fpexc), N);
    elsif altfp && op3_nan && op1_nan then                  // <m> register NaN selected
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type_nan, op3, fpcr, fpexc), N);
    elsif type1 == FPType_SNaN then
        done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_SNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr, fpexc), N);
    elsif type3 == FPType_SNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr, fpexc), N);
    elsif type1 == FPType_QNaN then
        done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_QNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr, fpexc), N);
    elsif type3 == FPType_QNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr, fpexc), N);
    else
        done = FALSE; result = Zeros(N); // 'Don't care' result
    return (done, result);
```

Library pseudocode for shared/functions/float/fpmulx/FPMulX

```
// FPMulX()
// ========

bits(N) FPMulX(bits(N) op1, bits(N) op2, FPCRType fpcr)

    assert N IN {16,32,64};
    bits(N) result;
    boolean done;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo(sign1 EOR sign2, N);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, N);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2, N);
        else
            result = FPRound(value1*value2, fpcr, N);

        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpneg/FPNeg

```
// FPNeg()
// =======

bits(N) FPNeg(bits(N) op)

    assert N IN {16,32,64};
    if !UsingAArch32() && HaveAltFP() then
        FPCRType fpcr = FPCR[];
        if fpcr.AH == '1' then
            (fptype, -, -) = FPUnpack(op, fpcr, FALSE);
            if fptype IN {FPType_SNaN, FPType_QNaN} then

                return op;        // When fpcr.AH=1, sign of NaN has no consequence

    return NOT(op<N-1>) : op<N-2:0>;
```

Library pseudocode for shared/functions/float/fponepointfive/FPOnePointFive

```
// FPOnePointFive()
// ================

bits(N) FPOnePointFive(bit sign, integer N)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp  = '0':Ones(E-1);
    frac = '1':Zeros(F-1);
    result = sign : exp : frac;

    return result;
```

Library pseudocode for shared/functions/float/fpprocessdenorms/FPProcessDenorm

```
// FPProcessDenorm()
// =================
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenorm(FPType fptype, integer N, FPCRType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && fptype == FPType_Denormal then
        FPProcessException(FPExc_InputDenorm, fpcr);
```

Library pseudocode for shared/functions/float/fpprocessdenorms/FPProcessDenorms

```
// FPProcessDenorms()
// ==================
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenorms(FPType type1, FPType type2, integer N, FPCRType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPType_Denormal || type2 == FPType_Denormal) then
        FPProcessException(FPExc_InputDenorm, fpcr);
```

Library pseudocode for shared/functions/float/fpprocessdenorms/FPProcessDenorms3

```
// FPProcessDenorms3()
// ===================
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenorms3(FPType type1, FPType type2, FPType type3, integer N, FPCRType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPType_Denormal || type2 == FPType_Denormal ||
        type3 == FPType_Denormal) then
        FPProcessException(FPExc_InputDenorm, fpcr);
```

Library pseudocode for shared/functions/float/fpprocessdenorms/FPProcessDenorms4

```
// FPProcessDenorms4()
// ===================
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenorms4(FPType type1, FPType type2, FPType type3, FPType type4, integer N, FPCRType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPType_Denormal || type2 == FPType_Denormal ||
        type3 == FPType_Denormal || type4 == FPType_Denormal) then
        FPProcessException(FPExc_InputDenorm, fpcr);
```

Library pseudocode for shared/functions/float/fpprocessexception/FPProcessException

```
// FPProcessException()
// ====================
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

FPProcessException(FPExc exception, FPCRType fpcr)

    integer cumul;
    // Determine the cumulative exception bit number
    case exception of
        when FPExc_InvalidOp     cumul = 0;
        when FPExc_DivideByZero  cumul = 1;
        when FPExc_Overflow      cumul = 2;
        when FPExc_Underflow     cumul = 3;
        when FPExc_Inexact       cumul = 4;
        when FPExc_InputDenorm   cumul = 7;
    enable = cumul + 8;
    if fpcr<enable> == '1' && (!HaveSME() || PSTATE.SM == '0' || IsFullA64Enabled()) then
        // Trapping of the exception enabled.
        // It is IMPLEMENTATION DEFINED whether the enable bit may be set at all,
        // and if so then how exceptions and in what order that they may be
        // accumulated before calling FPTrappedException().
        bits(8) accumulated_exceptions = GetAccumulatedFPExceptions();
        accumulated_exceptions<cumul> = '1';
        if boolean IMPLEMENTATION_DEFINED "Support trapping of floating-point exceptions" then
            if UsingAArch32() then
                AArch32.FPTrappedException(accumulated_exceptions);
            else
                is_ase = IsASEInstruction();
                AArch64.FPTrappedException(is_ase, accumulated_exceptions);
        else
            // The exceptions generated by this instruction are accumulated by the PE and
            // FPTrappedException is called later during its execution, before the next
            // instruction is executed. This field is cleared at the start of each FP instruction.
            SetAccumulatedFPExceptions(accumulated_exceptions);
    elsif UsingAArch32() then
        // Set the cumulative exception bit
        FPSCR<cumul> = '1';
    else
        // Set the cumulative exception bit
        FPSR<cumul> = '1';

    return;
```

Library pseudocode for shared/functions/float/fpprocessnan/FPProcessNaN

```
// FPProcessNaN()
// ==============

bits(N) FPProcessNaN(FPType fptype, bits(N) op, FPCRType fpcr)
    boolean fpexc = TRUE;   // Generate floating-point exceptions
    return FPProcessNaN(fptype, op, fpcr, fpexc);

// FPProcessNaN()
// ==============
// Handle NaN input operands, returning the operand or default NaN value
// if fpcr.DN is selected. The 'fpcr' argument supplies the FPCR control bits.
// The 'fpexc' argument controls the generation of exceptions, regardless of
// whether 'fptype' is a signalling NaN or a quiet NaN.

bits(N) FPProcessNaN(FPType fptype, bits(N) op, FPCRType fpcr, boolean fpexc)

    assert N IN {16,32,64};
    assert fptype IN {FPType_QNaN, FPType_SNaN};
    integer topfrac;

    case N of
        when 16 topfrac =  9;
        when 32 topfrac = 22;
        when 64 topfrac = 51;

    result = op;
    if fptype == FPType_SNaN then
        result<topfrac> = '1';
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
    if fpcr.DN == '1' then  // DefaultNaN requested
        result = FPDefaultNaN(fpcr, N);

    return result;
```

Library pseudocode for shared/functions/float/fpprocessnans/FPProcessNaNs

```
// FPProcessNaNs()
// ===============

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2, bits(N) op1,
                                 bits(N) op2, FPCRType fpcr)
    boolean fpexc       = TRUE;     // Generate floating-point exceptions
    return FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);

// FPProcessNaNs()
// ===============
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits and 'altfmaxfmin' controls
// alternative floating-point behaviour for FMAX, FMIN and variants. 'fpexc'
// controls the generation of floating-point exceptions. Status information
// is updated directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2, bits(N) op1, bits(N) op2,
                                 FPCRType fpcr, boolean fpexc)

    assert N IN {16,32,64};
    boolean done;
    bits(N) result;
    boolean altfp    = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    boolean op1_nan  = type1 IN {FPType_SNaN, FPType_QNaN};
    boolean op2_nan  = type2 IN {FPType_SNaN, FPType_QNaN};
    boolean any_snan = type1 == FPType_SNaN || type2 == FPType_SNaN;
    FPType  type_nan = if any_snan then FPType_SNaN else FPType_QNaN;

    if altfp && op1_nan && op2_nan then
        // <n> register NaN selected
        done = TRUE;  result = FPProcessNaN(type_nan, op1, fpcr, fpexc);
    elsif type1 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr, fpexc);
    elsif type1 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr, fpexc);
    else
        done = FALSE;  result = Zeros(N);  // 'Don't care' result

    return (done, result);
```

Library pseudocode for shared/functions/float/fpprocessnans3/FPProcessNaNs3

```
// FPProcessNaNs3()
// ================

(boolean, bits(N)) FPProcessNaNs3(FPType type1, FPType type2, FPType type3,
                                  bits(N) op1, bits(N) op2, bits(N) op3,
                                  FPCRType fpcr)
    boolean fpexc = TRUE;   // Generate floating-point exceptions
    return FPProcessNaNs3(type1, type2, type3, op1, op2, op3, fpcr, fpexc);

// FPProcessNaNs3()
// ================
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs3(FPType type1, FPType type2, FPType type3,
                                  bits(N) op1, bits(N) op2, bits(N) op3,
                                  FPCRType fpcr, boolean fpexc)

    assert N IN {16,32,64};
    bits(N) result;
    boolean op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};
    boolean op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};
    boolean op3_nan = type3 IN {FPType_SNaN, FPType_QNaN};

    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    FPType type_nan;
    if altfp then
        if type1 == FPType_SNaN || type2 == FPType_SNaN || type3 == FPType_SNaN then
            type_nan = FPType_SNaN;
        else
            type_nan = FPType_QNaN;

    boolean done;
    if altfp && op1_nan && op2_nan && op3_nan then
        // <n> register NaN selected
        done = TRUE;  result = FPProcessNaN(type_nan, op2, fpcr, fpexc);
    elsif altfp && op2_nan && (op1_nan || op3_nan) then
        // <n> register NaN selected
        done = TRUE;  result = FPProcessNaN(type_nan, op2, fpcr, fpexc);
    elsif altfp && op3_nan && op1_nan then
        // <m> register NaN selected
        done = TRUE;  result = FPProcessNaN(type_nan, op3, fpcr, fpexc);
    elsif type1 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr, fpexc);
    elsif type3 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type3, op3, fpcr, fpexc);
    elsif type1 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr, fpexc);
    elsif type3 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type3, op3, fpcr, fpexc);
    else
        done = FALSE;  result = Zeros(N);  // 'Don't care' result

    return (done, result);
```

Library pseudocode for shared/functions/float/fpprocessnans4/FPProcessNaNs4

```
// FPProcessNaNs4()
// ================
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits.
// Status information is updated directly in the FPSR where appropriate.
// The 'fpexc' controls the generation of floating-point exceptions.

(boolean, bits(N)) FPProcessNaNs4(FPType type1, FPType type2, FPType type3, FPType type4,
                                  bits(N DIV 2) op1, bits(N DIV 2) op2, bits(N DIV 2) op3,
                                  bits(N DIV 2) op4, FPCRType fpcr, boolean fpexc)

    assert N == 32;

    bits(N) result;
    boolean done;
    // The FPCR.AH control does not affect these checks
    if type1 == FPType_SNaN then
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type1, op1, fpcr, fpexc), N);
    elsif type2 == FPType_SNaN then
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr, fpexc), N);
    elsif type3 == FPType_SNaN then
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr, fpexc), N);
    elsif type4 == FPType_SNaN then
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type4, op4, fpcr, fpexc), N);
    elsif type1 == FPType_QNaN then
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type1, op1, fpcr, fpexc), N);
    elsif type2 == FPType_QNaN then
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr, fpexc), N);
    elsif type3 == FPType_QNaN then
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr, fpexc), N);
    elsif type4 == FPType_QNaN then
        done = TRUE;  result = FPConvertNaN(FPProcessNaN(type4, op4, fpcr, fpexc), N);
    else
        done = FALSE;  result = Zeros(N);  // 'Don't care' result

    return (done, result);
```

Library pseudocode for shared/functions/float/fprecipestimate/FPRecipEstimate

```
// FPRecipEstimate()
// =================

bits(N) FPRecipEstimate(bits(N) operand, FPCRType fpcr_in)

    assert N IN {16,32,64};
    FPCRType fpcr = fpcr_in;
    bits(N) result;
    boolean overflow_to_inf;
    // When using alternative floating-point behaviour, do not generate
    // floating-point exceptions, flush denormal input and output to zero,
    // and use RNE rounding mode.
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    boolean fpexc = !altfp;
    if altfp then fpcr.<FIZ,FZ> = '11';
    if altfp then fpcr.RMode    = '00';

    (fptype,sign,value) = FPUnpack(operand, fpcr, fpexc);

    FPRounding rounding = FPRoundingMode(fpcr);
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, operand, fpcr, fpexc);
    elsif fptype == FPType_Infinity then
        result = FPZero(sign, N);
    elsif fptype == FPType_Zero then
        result = FPInfinity(sign, N);
        if fpexc then FPProcessException(FPExc_DivideByZero, fpcr);
    elsif (
            (N == 16 && Abs(value) < 2.0^-16) ||
            (N == 32 && Abs(value) < 2.0^-128) ||
            (N == 64 && Abs(value) < 2.0^-1024)
          ) then
        case rounding of
            when FPRounding_TIEEVEN
                overflow_to_inf = TRUE;
            when FPRounding_POSINF
                overflow_to_inf = (sign == '0');
            when FPRounding_NEGINF
                overflow_to_inf = (sign == '1');
            when FPRounding_ZERO
                overflow_to_inf = FALSE;
        result = if overflow_to_inf then FPInfinity(sign, N) else FPMaxNormal(sign, N);
        if fpexc then
            FPProcessException(FPExc_Overflow, fpcr);
            FPProcessException(FPExc_Inexact, fpcr);
    elsif ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16))
          && (
                (N == 16 && Abs(value) >= 2.0^14) ||
                (N == 32 && Abs(value) >= 2.0^126) ||
                (N == 64 && Abs(value) >= 2.0^1022)
              ) then
        // Result flushed to zero of correct sign
        result = FPZero(sign, N);

        // Flush-to-zero never generates a trapped exception.
        if UsingAArch32() then
            FPSCR.UFC = '1';
        else
            if fpexc then FPSR.UFC = '1';
    else
        // Scale to a fixed point value in the range 0.5 <= x < 1.0 in steps of 1/512, and
        // calculate result exponent. Scaled value has copied sign bit,
        // exponent = 1022 = double-precision biased version of -1,
        // fraction = original fraction
        bits(52) fraction;
        integer exp;
        case N of
            when 16
                fraction = operand<9:0> : Zeros(42);
                exp = UInt(operand<14:10>);
```

```
            when 32
                fraction = operand<22:0> : Zeros(29);
                exp = UInt(operand<30:23>);
            when 64
                fraction = operand<51:0>;
                exp = UInt(operand<62:52>);

        if exp == 0 then
            if fraction<51> == '0' then
                exp = -1;
                fraction = fraction<49:0>:'00';
            else
                fraction = fraction<50:0>:'0';

        integer scaled;
        boolean increasedprecision = N==32 && HaveFeatRPRES() && altfp;

        if !increasedprecision then
            scaled = UInt('1':fraction<51:44>);
        else
            scaled = UInt('1':fraction<51:41>);

        integer result_exp;
        case N of
            when 16 result_exp =   29 - exp; // In range 29-30 = -1 to 29+1 = 30
            when 32 result_exp =  253 - exp; // In range 253-254 = -1 to 253+1 = 254
            when 64 result_exp = 2045 - exp; // In range 2045-2046 = -1 to 2045+1 = 2046

        // Scaled is in range 256 .. 511 or 2048 .. 4095 range representing a
        // fixed-point number in range [0.5 .. 1.0].
        estimate = RecipEstimate(scaled, increasedprecision);

        // Estimate is in the range 256 .. 511 or 4096 .. 8191 representing a
        // fixed-point result in the range [1.0 .. 2.0].
        // Convert to scaled floating point result with copied sign bit,
        // high-order bits from estimate, and exponent calculated above.
        if !increasedprecision then
            fraction = estimate<7:0> : Zeros(44);
        else
            fraction = estimate<11:0> : Zeros(40);

        if result_exp == 0 then
            fraction = '1' : fraction<51:1>;
        elsif result_exp == -1 then
            fraction = '01' : fraction<51:2>;
            result_exp = 0;

        case N of
            when 16 result = sign : result_exp<N-12:0> : fraction<51:42>;
            when 32 result = sign : result_exp<N-25:0> : fraction<51:29>;
            when 64 result = sign : result_exp<N-54:0> : fraction<51:0>;

    return result;
```

Library pseudocode for shared/functions/float/fprecipestimate/RecipEstimate

```
// RecipEstimate()
// ===============
// Compute estimate of reciprocal of 9-bit fixed-point number.
//
// a is in range 256 .. 511 or 2048 .. 4096 representing a number in
// the range 0.5 <= x < 1.0.
// increasedprecision determines if the mantissa is 8-bit or 12-bit.
// result is in the range 256 .. 511 or 4096 .. 8191 representing a
// number in the range 1.0 to 511/256 or 1.00 to 8191/4096.

integer RecipEstimate(integer a_in, boolean increasedprecision)

    integer a = a_in;
    integer r;
    if !increasedprecision then
        assert 256 <= a && a < 512;
        a = a*2+1;                      // Round to nearest
        integer b = (2 ^ 19) DIV a;
        r = (b+1) DIV 2;                // Round to nearest
        assert 256 <= r && r < 512;
    else
        assert 2048 <= a && a < 4096;
        a = a*2+1;                      // Round to nearest
        real real_val = Real(2^25)/Real(a);
        r = RoundDown(real_val);
        real error = real_val - Real(r);
        boolean round_up = error > 0.5;  // Error cannot be exactly 0.5 so do not need tie case
        if round_up then r = r+1;
        assert 4096 <= r && r < 8192;

    return r;
```

Library pseudocode for shared/functions/float/fprecpx/FPRecpX

```
// FPRecpX()
// =========

bits(N) FPRecpX(bits(N) op, FPCRType fpcr_in)

    assert N IN {16,32,64};
    FPCRType fpcr = fpcr_in;
    integer esize;
    case N of
        when 16 esize =  5;
        when 32 esize =  8;
        when 64 esize = 11;

    bits(N)           result;
    bits(esize)       exp;
    bits(esize)       max_exp;
    bits(N-(esize+1)) frac = Zeros(N-(esize+1));

    boolean altfp = HaveAltFP() && fpcr.AH == '1';
    boolean fpexc = !altfp;                 // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11';     // Flush denormal input and output to zero
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

    case N of
        when 16 exp = op<10+esize-1:10>;
        when 32 exp = op<23+esize-1:23>;
        when 64 exp = op<52+esize-1:52>;

    max_exp = Ones(esize) - 1;

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, op, fpcr, fpexc);
    else
        if IsZero(exp) then                 // Zero and denormals
            result = sign:max_exp:frac;
        else                                // Infinities and normals
            result = sign:NOT(exp):frac;

    return result;
```

Library pseudocode for shared/functions/float/fpround/FPRound

```
// FPRound()
// =========
// Generic conversion from precise, unbounded real data type to IEEE format.

bits(N) FPRound(real op, FPCRType fpcr, integer N)
    return FPRound(op, fpcr, FPRoundingMode(fpcr), N);

// FPRound()
// =========
// For directed FP conversion, includes an explicit 'rounding' argument.

bits(N) FPRound(real op, FPCRType fpcr_in, FPRounding rounding, integer N)
    boolean fpexc = TRUE;    // Generate floating-point exceptions
    return FPRound(op, fpcr_in, rounding, fpexc, N);

// FPRound()
// =========
// For AltFP, includes an explicit FPEXC argument to disable exception
// generation and switches off Arm alternate half-precision mode.

bits(N) FPRound(real op, FPCRType fpcr_in, FPRounding rounding, boolean fpexc, integer N)
    FPCRType fpcr = fpcr_in;
    fpcr.AHP = '0';
    boolean isbfloat16 = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc, N);
```

Library pseudocode for shared/functions/float/fpround/FPRoundBase

```
// FPRoundBase()
// =============
// For BFloat16, includes an explicit 'isbfloat16' argument.

bits(N) FPRoundBase(real op, FPCRType fpcr, FPRounding rounding, boolean isbfloat16, integer N)
    boolean fpexc = TRUE;    // Generate floating-point exceptions
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc, N);

// FPRoundBase()
// =============
// Convert a real number 'op' into an N-bit floating-point value using the
// supplied rounding mode 'rounding'.
//
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

bits(N) FPRoundBase(real op, FPCRType fpcr, FPRounding rounding,
                    boolean isbfloat16, boolean fpexc, integer N)

    assert N IN {16,32,64};
    assert op != 0.0;
    assert rounding != FPRounding_TIEAWAY;
    bits(N) result;

    // Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
    integer minimum_exp;
    integer F;
    integer E;
    if N == 16 then
        minimum_exp = -14;  E = 5;  F = 10;
    elsif N == 32 && isbfloat16 then
        minimum_exp = -126;  E = 8;  F = 7;
    elsif N == 32 then
        minimum_exp = -126;  E = 8;  F = 23;
    else  // N == 64
        minimum_exp = -1022;  E = 11;  F = 52;

    // Split value into sign, unrounded mantissa and exponent.
    bit sign;
    real mantissa;
    if op < 0.0 then
        sign = '1';  mantissa = -op;
    else
        sign = '0';  mantissa = op;
    exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0;  exponent = exponent - 1;
    while mantissa >= 2.0 do
        mantissa = mantissa / 2.0;  exponent = exponent + 1;

    // When TRUE, detection of underflow occurs after rounding and the test for a
    // denormalized number for single and double precision values occurs after rounding.
    altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';

    // Deal with flush-to-zero before rounding if FPCR.AH != '1'.
    if (!altfp && ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16)) &&
        exponent < minimum_exp) then
        // Flush-to-zero never generates a trapped exception.
        if UsingAArch32() then
            FPSCR.UFC = '1';
        else
            if fpexc then FPSR.UFC = '1';
        return FPZero(sign, N);

    biased_exp_unconstrained = (exponent - minimum_exp) + 1;
    int_mant_unconstrained = RoundDown(mantissa * 2.0^F);
    error_unconstrained = mantissa * 2.0^F - Real(int_mant_unconstrained);

    // Start creating the exponent value for the result. Start by biasing the actual exponent
```

```
// so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
biased_exp = Max((exponent - minimum_exp) + 1, 0);
if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

// Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
int_mant = RoundDown(mantissa * 2.0^F);  // < 2.0^F if biased_exp == 0, >= 2.0^F if not
error = mantissa * 2.0^F - Real(int_mant);

// Underflow occurs if exponent is too small before rounding, and result is inexact or
// the Underflow exception is trapped. This applies before rounding if FPCR.AH != '1'.
boolean trapped_UF = fpcr.UFE == '1' && (!InStreamingMode() || IsFullA64Enabled());
if !altfp && biased_exp == 0 && (error != 0.0 || trapped_UF) then
    if fpexc then FPProcessException(FPExc_Underflow, fpcr);

// Round result according to rounding mode.
boolean round_up_unconstrained;
boolean round_up;
boolean overflow_to_inf;
if altfp then

    case rounding of
        when FPRounding_TIEEVEN
            round_up_unconstrained = (error_unconstrained > 0.5 ||
               (error_unconstrained == 0.5 && int_mant_unconstrained<0> == '1'));
            round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
            overflow_to_inf = TRUE;
        when FPRounding_POSINF
            round_up_unconstrained = (error_unconstrained != 0.0 && sign == '0');
            round_up = (error != 0.0 && sign == '0');
            overflow_to_inf = (sign == '0');
        when FPRounding_NEGINF
            round_up_unconstrained = (error_unconstrained != 0.0 && sign == '1');
            round_up = (error != 0.0 && sign == '1');
            overflow_to_inf = (sign == '1');
        when FPRounding_ZERO, FPRounding_ODD
            round_up_unconstrained = FALSE;
            round_up = FALSE;
            overflow_to_inf = FALSE;

    if round_up_unconstrained then
        int_mant_unconstrained = int_mant_unconstrained + 1;
        if int_mant_unconstrained == 2^(F+1) then    // Rounded up to next exponent
            biased_exp_unconstrained = biased_exp_unconstrained + 1;
            int_mant_unconstrained   = int_mant_unconstrained DIV 2;

    // Deal with flush-to-zero and underflow after rounding if FPCR.AH == '1'.
    if biased_exp_unconstrained < 1 && int_mant_unconstrained != 0 then
        // the result of unconstrained rounding is less than the minimum normalized number
        if (fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16) then   // Flush-to-zero
            if fpexc then
                FPSR.UFC = '1';
                FPProcessException(FPExc_Inexact, fpcr);
            return FPZero(sign, N);
        elsif error != 0.0 || trapped_UF then
            if fpexc then FPProcessException(FPExc_Underflow, fpcr);
else    // altfp == FALSE
    case rounding of
        when FPRounding_TIEEVEN
            round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
            overflow_to_inf = TRUE;
        when FPRounding_POSINF
            round_up = (error != 0.0 && sign == '0');
            overflow_to_inf = (sign == '0');
        when FPRounding_NEGINF
            round_up = (error != 0.0 && sign == '1');
            overflow_to_inf = (sign == '1');
        when FPRounding_ZERO, FPRounding_ODD
            round_up = FALSE;
            overflow_to_inf = FALSE;
```

```
    if round_up then
        int_mant = int_mant + 1;
        if int_mant == 2^F then      // Rounded up from denormalized to normalized
            biased_exp = 1;
        if int_mant == 2^(F+1) then  // Rounded up to next exponent
            biased_exp = biased_exp + 1;
            int_mant = int_mant DIV 2;

    // Handle rounding to odd
    if error != 0.0 && rounding == FPRounding_ODD then
        int_mant<0> = '1';

    // Deal with overflow and generate result.
    if N != 16 || fpcr.AHP == '0' then  // Single, double or IEEE half precision
        if biased_exp >= 2^E - 1 then
            result = if overflow_to_inf then FPInfinity(sign, N) else FPMaxNormal(sign, N);
            if fpexc then FPProcessException(FPExc_Overflow, fpcr);
            error = 1.0;  // Ensure that an Inexact exception occurs
        else
            result = sign : biased_exp<E-1:0> : int_mant<F-1:0> : Zeros(N-(E+F+1));
    else                                    // Alternative half precision
        if biased_exp >= 2^E then
            result = sign : Ones(N-1);
            if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
            error = 0.0;  // Ensure that an Inexact exception does not occur
        else
            result = sign : biased_exp<E-1:0> : int_mant<F-1:0> : Zeros(N-(E+F+1));

    // Deal with Inexact exception.
    if error != 0.0 then
        if fpexc then FPProcessException(FPExc_Inexact, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpround/FPRoundCV

```
// FPRoundCV()
// ===========
// Used for FP to FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

bits(N) FPRoundCV(real op, FPCRType fpcr_in, FPRounding rounding, integer N)
    FPCRType fpcr = fpcr_in;
    fpcr.FZ16 = '0';
    boolean fpexc = TRUE;    // Generate floating-point exceptions
    boolean isbfloat16 = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc, N);
```

Library pseudocode for shared/functions/float/fprounding/FPRounding

```
enumeration FPRounding  {FPRounding_TIEEVEN, FPRounding_POSINF,
                         FPRounding_NEGINF,  FPRounding_ZERO,
                         FPRounding_TIEAWAY, FPRounding_ODD};
```

Library pseudocode for shared/functions/float/fproundingmode/FPRoundingMode

```
// FPRoundingMode()
// ================
// Return the current floating-point rounding mode.

FPRounding FPRoundingMode(FPCRType fpcr)
    return FPDecodeRounding(fpcr.RMode);
```

Library pseudocode for shared/functions/float/fproundint/FPRoundInt

```
// FPRoundInt()
// ============

// Round op to nearest integral floating point value using rounding mode in FPCR/FPSCR.
// If EXACT is TRUE, set FPSR.IXC if result is not numerically equal to op.

bits(N) FPRoundInt(bits(N) op, FPCRType fpcr, FPRounding rounding, boolean exact)

    assert rounding != FPRounding_ODD;
    assert N IN {16,32,64};

    // When alternative floating-point support is TRUE, do not generate
    // Input Denormal floating-point exceptions.
    altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    fpexc = !altfp;

    // Unpack using FPCR to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

    bits(N) result;
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elsif fptype == FPType_Infinity then
        result = FPInfinity(sign, N);
    elsif fptype == FPType_Zero then
        result = FPZero(sign, N);
    else
        // Extract integer component.
        int_result = RoundDown(value);
        error = value - Real(int_result);

        // Determine whether supplied rounding mode requires an increment.
        boolean round_up;
        case rounding of
            when FPRounding_TIEEVEN
                round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
            when FPRounding_POSINF
                round_up = (error != 0.0);
            when FPRounding_NEGINF
                round_up = FALSE;
            when FPRounding_ZERO
                round_up = (error != 0.0 && int_result < 0);
            when FPRounding_TIEAWAY
                round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

        if round_up then int_result = int_result + 1;

        // Convert integer value into an equivalent real value.
        real_result = Real(int_result);

        // Re-encode as a floating-point value, result is always exact.
        if real_result == 0.0 then
            result = FPZero(sign, N);
        else
            result = FPRound(real_result, fpcr, FPRounding_ZERO, N);

        // Generate inexact exceptions.
        if error != 0.0 && exact then
            FPProcessException(FPExc_Inexact, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fproundintn/FPRoundIntN

```
// FPRoundIntN()
// =============

bits(N) FPRoundIntN(bits(N) op, FPCRType fpcr, FPRounding rounding, integer intsize)
    assert rounding != FPRounding_ODD;
    assert N IN {32,64};
    assert intsize IN {32, 64};
    integer exp;
    bits(N) result;
    boolean round_up;
    constant integer E = (if N == 32 then 8 else 11);
    constant integer F = N - (E + 1);

    // When alternative floating-point support is TRUE, do not generate
    // Input Denormal floating-point exceptions.
    altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    fpexc = !altfp;

    // Unpack using FPCR to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

    if fptype IN {FPType_SNaN, FPType_QNaN, FPType_Infinity} then
        if N == 32 then
            exp = 126 + intsize;
            result = '1':exp<(E-1):0>:Zeros(F);
        else
            exp = 1022+intsize;
            result = '1':exp<(E-1):0>:Zeros(F);
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif fptype == FPType_Zero then
        result = FPZero(sign, N);
    else
        // Extract integer component.
        int_result = RoundDown(value);
        error = value - Real(int_result);

        // Determine whether supplied rounding mode requires an increment.
        case rounding of
            when FPRounding_TIEEVEN
                round_up = error > 0.5 || (error == 0.5 && int_result<0> == '1');
            when FPRounding_POSINF
                round_up = error != 0.0;
            when FPRounding_NEGINF
                round_up = FALSE;
            when FPRounding_ZERO
                round_up = error != 0.0 && int_result < 0;
            when FPRounding_TIEAWAY
                round_up = error > 0.5 || (error == 0.5 && int_result >= 0);

        if round_up then int_result = int_result + 1;
        overflow = int_result > 2^(intsize-1)-1 || int_result < -1*2^(intsize-1);

        if overflow then
            if N == 32 then
                exp = 126 + intsize;
                result = '1':exp<(E-1):0>:Zeros(F);
            else
                exp = 1022 + intsize;
                result = '1':exp<(E-1):0>:Zeros(F);
            FPProcessException(FPExc_InvalidOp, fpcr);
            // This case shouldn't set Inexact.
            error = 0.0;

        else
            // Convert integer value into an equivalent real value.
            real_result = Real(int_result);

            // Re-encode as a floating-point value, result is always exact.
            if real_result == 0.0 then
                result = FPZero(sign, N);
```

```
            else
                result = FPRound(real_result, fpcr, FPRounding_ZERO, N);

        // Generate inexact exceptions.
        if error != 0.0 then
            FPProcessException(FPExc_Inexact, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fprsqrtestimate/FPRSqrtEstimate

```
// FPRSqrtEstimate()
// =================

bits(N) FPRSqrtEstimate(bits(N) operand, FPCRType fpcr_in)

    assert N IN {16,32,64};
    FPCRType fpcr = fpcr_in;

    // When using alternative floating-point behaviour, do not generate
    // floating-point exceptions and flush denormal input to zero.
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    boolean fpexc = !altfp;
    if altfp then fpcr.<FIZ,FZ> = '11';

    (fptype,sign,value) = FPUnpack(operand, fpcr, fpexc);

    bits(N) result;
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, operand, fpcr, fpexc);
    elsif fptype == FPType_Zero then
        result = FPInfinity(sign, N);
        if fpexc then FPProcessException(FPExc_DivideByZero, fpcr);
    elsif sign == '1' then
        result = FPDefaultNaN(fpcr, N);
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
    elsif fptype == FPType_Infinity then
        result = FPZero('0', N);
    else
        // Scale to a fixed-point value in the range 0.25 <= x < 1.0 in steps of 512, with the
        // evenness or oddness of the exponent unchanged, and calculate result exponent.
        // Scaled value has copied sign bit, exponent = 1022 or 1021 = double-precision
        // biased version of -1 or -2, fraction = original fraction extended with zeros.

        bits(52) fraction;
        integer exp;
        case N of
            when 16
                fraction = operand<9:0> : Zeros(42);
                exp = UInt(operand<14:10>);
            when 32
                fraction = operand<22:0> : Zeros(29);
                exp = UInt(operand<30:23>);
            when 64
                fraction = operand<51:0>;
                exp = UInt(operand<62:52>);

        if exp == 0 then
            while fraction<51> == '0' do
                fraction = fraction<50:0> : '0';
                exp = exp - 1;
            fraction = fraction<50:0> : '0';

        integer scaled;
        boolean increasedprecision = N==32 && HaveFeatRPRES() && altfp;

        if !increasedprecision then
            if exp<0> == '0' then
                scaled = UInt('1':fraction<51:44>);
            else
                scaled = UInt('01':fraction<51:45>);
        else
            if exp<0> == '0' then
                scaled = UInt('1':fraction<51:41>);
            else
                scaled = UInt('01':fraction<51:42>);

        integer result_exp;
        case N of
            when 16 result_exp = (  44 - exp) DIV 2;
            when 32 result_exp = ( 380 - exp) DIV 2;
```

```
            when 64 result_exp = (3068 - exp) DIV 2;

        estimate = RecipSqrtEstimate(scaled, increasedprecision);

        // Estimate is in the range 256 .. 511 or 4096 .. 8191 representing a
        // fixed-point result in the range [1.0 .. 2.0].
        // Convert to scaled floating point result with copied sign bit and high-order
        // fraction bits, and exponent calculated above.
        case N of
            when 16 result = '0' : result_exp<N-12:0> : estimate<7:0>:Zeros(2);
            when 32
                if !increasedprecision then
                    result = '0' : result_exp<N-25:0> : estimate<7:0>:Zeros(15);
                else
                    result = '0' : result_exp<N-25:0> : estimate<11:0>:Zeros(11);
            when 64 result = '0' : result_exp<N-54:0> : estimate<7:0>:Zeros(44);

    return result;
```

Library pseudocode for shared/functions/float/fprsqrtestimate/RecipSqrtEstimate

```
// RecipSqrtEstimate()
// ===================
// Compute estimate of reciprocal square root of 9-bit fixed-point number.
//
// a_in is in range 128 .. 511 or 1024 .. 4095, with increased precision,
// representing a number in the range 0.25 <= x < 1.0.
// increasedprecision determines if the mantissa is 8-bit or 12-bit.
// result is in the range 256 .. 511 or 4096 .. 8191, with increased precision,
// representing a number in the range 1.0 to 511/256 or 8191/4096.

integer RecipSqrtEstimate(integer a_in, boolean increasedprecision)

    integer a = a_in;
    integer r;
    if !increasedprecision then
        assert 128 <= a && a < 512;
        if a < 256 then                     // 0.25 .. 0.5
            a = a*2+1;                      // a in units of 1/512 rounded to nearest
        else                                // 0.5 .. 1.0
            a = (a >> 1) << 1;              // Discard bottom bit
            a = (a+1)*2;                    // a in units of 1/256 rounded to nearest
        integer b = 512;
        while a*(b+1)*(b+1) < 2^28 do
            b = b+1;
        // b = largest b such that b < 2^14 / sqrt(a)
        r = (b+1) DIV 2;                    // Round to nearest
        assert 256 <= r && r < 512;
    else
        assert 1024 <= a && a < 4096;
        real real_val;
        real error;
        integer int_val;

        if a < 2048 then                    // 0.25... 0.5
            a = a*2 + 1;                    // Take 10 bits of fraction and force a 1 at the bottom
            real_val = Real(a)/2.0;
        else                                // 0.5..1.0
            a = (a >> 1) << 1;              // Discard bottom bit
            a = a+1;                        // Take 10 bits of fraction and force a 1 at the bottom
            real_val = Real(a);

        real_val = Sqrt(real_val);          // This number will lie in the range of 32 to 64
                                            // Round to nearest even for a DP float number
        real_val = real_val * Real(2^47);   // The integer is the size of the whole DP mantissa
        int_val  = RoundDown(real_val);     // Calculate rounding value
        error    = real_val - Real(int_val);
        round_up = error > 0.5;             // Error cannot be exactly 0.5 so do not need tie case
        if round_up then int_val = int_val+1;

        real_val = Real(2^65)/Real(int_val); // Lies in the range 4096 <= real_val < 8192
        int_val  = RoundDown(real_val);     // Round that (to nearest even) to give integer
        error    = real_val - Real(int_val);
        round_up = (error > 0.5 || (error == 0.5 && int_val<0> == '1'));
        if round_up then int_val = int_val+1;

        r = int_val;
        assert 4096 <= r && r < 8192;

    return r;
```

Library pseudocode for shared/functions/float/fpsqrt/FPSqrt

```
// FPSqrt()
// ========

bits(N) FPSqrt(bits(N) op, FPCRType fpcr)

    assert N IN {16,32,64};
    (fptype,sign,value) = FPUnpack(op, fpcr);

    bits(N) result;
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elsif fptype == FPType_Zero then
        result = FPZero(sign, N);
    elsif fptype == FPType_Infinity && sign == '0' then
        result = FPInfinity(sign, N);
    elsif sign == '1' then
        result = FPDefaultNaN(fpcr, N);
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        result = FPRound(Sqrt(value), fpcr, N);
        FPProcessDenorm(fptype, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpsub/FPSub

```
// FPSub()
// =======

bits(N) FPSub(bits(N) op1, bits(N) op2, FPCRType fpcr)

    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);

    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);

        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN(fpcr, N);
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0', N);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1', N);
        elsif zero1 && zero2 && sign1 == NOT(sign2) then
            result = FPZero(sign1, N);
        else
            result_value = value1 - value2;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, N);

        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpthree/FPThree

```
// FPThree()
// =========

bits(N) FPThree(bit sign, integer N)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp  = '1':Zeros(E-1);
    frac = '1':Zeros(F-1);
    result = sign : exp : frac;

    return result;
```

Library pseudocode for shared/functions/float/fptofixed/FPToFixed

```
// FPToFixed()
// ===========

// Convert N-bit precision floating point 'op' to M-bit fixed point with
// FBITS fractional bits, controlled by UNSIGNED and ROUNDING.

bits(M) FPToFixed(bits(N) op, integer fbits, boolean unsigned, FPCRType fpcr, FPRounding rounding, intege

    assert N IN {16,32,64};
    assert M IN {16,32,64};
    assert fbits >= 0;
    assert rounding != FPRounding_ODD;

    // When alternative floating-point support is TRUE, do not generate
    // Input Denormal floating-point exceptions.
    altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    fpexc = !altfp;

    // Unpack using fpcr to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

    // If NaN, set cumulative flag or take exception.
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);

    // Scale by fractional bits and produce integer rounded towards minus-infinity.
    value = value * 2.0^fbits;
    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment.
    boolean round_up;
    case rounding of
        when FPRounding_TIEEVEN
            round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
        when FPRounding_POSINF
            round_up = (error != 0.0);
        when FPRounding_NEGINF
            round_up = FALSE;
        when FPRounding_ZERO
            round_up = (error != 0.0 && int_result < 0);
        when FPRounding_TIEAWAY
            round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

    if round_up then int_result = int_result + 1;

    // Generate saturated result and exceptions.
    (result, overflow) = SatQ(int_result, M, unsigned);
    if overflow then
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fptofixedjs/FPToFixedJS

```
// FPToFixedJS()
// =============

// Converts a double precision floating point input value
// to a signed integer, with rounding to zero.

(bits(N), bit) FPToFixedJS(bits(M) op, FPCRType fpcr, boolean Is64, integer N)

    assert M == 64 && N == 32;

    // If FALSE, never generate Input Denormal floating-point exceptions.
    fpexc_idenorm = !(HaveAltFP() && !UsingAArch32() && fpcr.AH == '1');

    // Unpack using fpcr to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc_idenorm);

    z = '1';
    // If NaN, set cumulative flag or take exception.
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);
        z = '0';

    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment.

    round_it_up = (error != 0.0 && int_result < 0);
    if round_it_up then int_result = int_result + 1;

    integer result;
    if int_result < 0 then
        result = int_result - 2^32*RoundUp(Real(int_result)/Real(2^32));
    else
        result = int_result - 2^32*RoundDown(Real(int_result)/Real(2^32));

    // Generate exceptions.
    if int_result < -(2^31) || int_result > (2^31)-1 then
        FPProcessException(FPExc_InvalidOp, fpcr);
        z = '0';
    elsif error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);
        z = '0';
    elsif sign == '1' && value == 0.0 then
        z = '0';
    elsif sign == '0' && value == 0.0 && !IsZero(op<51:0>) then
        z = '0';

    if fptype == FPType_Infinity then result = 0;

    return (result<N-1:0>, z);
```

Library pseudocode for shared/functions/float/fptwo/FPTwo

```
// FPTwo()
// =======

bits(N) FPTwo(bit sign, integer N)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp  = '1':Zeros(E-1);
    frac = Zeros(F);
    result = sign : exp : frac;

    return result;
```

Library pseudocode for shared/functions/float/fptype/FPType

```
enumeration FPType {FPType_Zero,
                    FPType_Denormal,
                    FPType_Nonzero,
                    FPType_Infinity,
                    FPType_QNaN,
                    FPType_SNaN};
```

Library pseudocode for shared/functions/float/fpunpack/FPUnpack

```
// FPUnpack()
// ==========

(FPType, bit, real) FPUnpack(bits(N) fpval, FPCRType fpcr_in)
    FPCRType fpcr = fpcr_in;
    fpcr.AHP = '0';
    boolean fpexc = TRUE;   // Generate floating-point exceptions
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
    return (fp_type, sign, value);

// FPUnpack()
// ==========
//
// Used by data processing, int/fixed to FP and FP to int/fixed conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

(FPType, bit, real) FPUnpack(bits(N) fpval, FPCRType fpcr_in, boolean fpexc)
    FPCRType fpcr = fpcr_in;
    fpcr.AHP = '0';
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
    return (fp_type, sign, value);
```

Library pseudocode for shared/functions/float/fpunpack/FPUnpackBase

```
// FPUnpackBase()
// ==============

(FPType, bit, real) FPUnpackBase(bits(N) fpval, FPCRType fpcr, boolean fpexc)
    boolean isbfloat16 = FALSE;
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc, isbfloat16);
    return (fp_type, sign, value);

// FPUnpackBase()
// ==============
//
// Unpack a floating-point number into its type, sign bit and the real number
// that it represents. The real number result has the correct sign for numbers
// and infinities, is very large in magnitude for infinities, and is 0.0 for
// NaNs. (These values are chosen to simplify the description of comparisons
// and conversions.)
//
// The 'fpcr_in' argument supplies FPCR control bits, 'fpexc' controls the
// generation of floating-point exceptions and 'isbfloat16' determines whether
// N=16 signifies BFloat16 or half-precision type. Status information is updated
// directly in the FPSR where appropriate.

(FPType, bit, real) FPUnpackBase(bits(N) fpval, FPCRType fpcr_in, boolean fpexc,
                                 boolean isbfloat16)

    assert N IN {16,32,64};

    FPCRType fpcr = fpcr_in;

    boolean altfp = HaveAltFP() && !UsingAArch32();
    boolean fiz   = altfp && fpcr.FIZ == '1';
    boolean fz    = fpcr.FZ == '1' && !(altfp && fpcr.AH == '1');
    real value;
    bit sign;
    FPType fptype;

    if N == 16 && !isbfloat16 then
        sign   = fpval<15>;
        exp16  = fpval<14:10>;
        frac16 = fpval<9:0>;
        if IsZero(exp16) then
            if IsZero(frac16) || fpcr.FZ16 == '1' then
                fptype = FPType_Zero;  value = 0.0;
            else
                fptype = FPType_Denormal;  value = 2.0^-14 * (Real(UInt(frac16)) * 2.0^-10);
        elsif IsOnes(exp16) && fpcr.AHP == '0' then  // Infinity or NaN in IEEE format
            if IsZero(frac16) then
                fptype = FPType_Infinity;  value = 2.0^1000000;
            else
                fptype = if frac16<9> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            fptype = FPType_Nonzero;
            value = 2.0^(UInt(exp16)-15) * (1.0 + Real(UInt(frac16)) * 2.0^-10);

    elsif N == 32 || isbfloat16 then
        bits(8) exp32;
        bits(23) frac32;
        if isbfloat16 then
            sign   = fpval<15>;
            exp32  = fpval<14:7>;
            frac32 = fpval<6:0> : Zeros(16);
        else
            sign   = fpval<31>;
            exp32  = fpval<30:23>;
            frac32 = fpval<22:0>;

        if IsZero(exp32) then
            if IsZero(frac32) then
                // Produce zero if value is zero.
```

```
                fptype = FPType_Zero;  value = 0.0;
            elsif fz || fiz then        // Flush-to-zero if FIZ==1 or AH,FZ==01
                fptype = FPType_Zero;  value = 0.0;
                // Check whether to raise Input Denormal floating-point exception.
                // fpcr.FIZ==1 does not raise Input Denormal exception.
                if fz then
                    // Denormalized input flushed to zero
                    if fpexc then FPProcessException(FPExc_InputDenorm, fpcr);
            else
                fptype = FPType_Denormal;  value = 2.0^-126 * (Real(UInt(frac32)) * 2.0^-23);
        elsif IsOnes(exp32) then
            if IsZero(frac32) then
                fptype = FPType_Infinity;  value = 2.0^1000000;
            else
                fptype = if frac32<22> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            fptype = FPType_Nonzero;
            value = 2.0^(UInt(exp32)-127) * (1.0 + Real(UInt(frac32)) * 2.0^-23);

    else // N == 64
        sign   = fpval<63>;
        exp64  = fpval<62:52>;
        frac64 = fpval<51:0>;

        if IsZero(exp64) then
            if IsZero(frac64) then
                // Produce zero if value is zero.
                fptype = FPType_Zero;  value = 0.0;
            elsif fz || fiz then        // Flush-to-zero if FIZ==1 or AH,FZ==01
                fptype = FPType_Zero;  value = 0.0;
                // Check whether to raise Input Denormal floating-point exception.
                // fpcr.FIZ==1 does not raise Input Denormal exception.
                if fz then
                    // Denormalized input flushed to zero
                    if fpexc then FPProcessException(FPExc_InputDenorm, fpcr);
            else
                fptype = FPType_Denormal;  value = 2.0^-1022 * (Real(UInt(frac64)) * 2.0^-52);
        elsif IsOnes(exp64) then
            if IsZero(frac64) then
                fptype = FPType_Infinity;  value = 2.0^1000000;
            else
                fptype = if frac64<51> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            fptype = FPType_Nonzero;
            value = 2.0^(UInt(exp64)-1023) * (1.0 + Real(UInt(frac64)) * 2.0^-52);

    if sign == '1' then value = -value;

    return (fptype, sign, value);
```

Library pseudocode for shared/functions/float/fpunpack/FPUnpackCV

```
// FPUnpackCV()
// ============
//
// Used for FP to FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

(FPType, bit, real) FPUnpackCV(bits(N) fpval, FPCRType fpcr_in)
    FPCRType fpcr = fpcr_in;
    fpcr.FZ16 = '0';
    boolean fpexc = TRUE;   // Generate floating-point exceptions
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
    return (fp_type, sign, value);
```

Library pseudocode for shared/functions/float/fpzero/FPZero

```
// FPZero()
// ========

bits(N) FPZero(bit sign, integer N)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp  = Zeros(E);
    frac = Zeros(F);
    result = sign : exp : frac;

    return result;
```

Library pseudocode for shared/functions/float/vfpexpandimm/VFPExpandImm

```
// VFPExpandImm()
// ==============

bits(N) VFPExpandImm(bits(8) imm8, integer N)

    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = (N - E) - 1;
    sign = imm8<7>;
    exp  = NOT(imm8<6>):Replicate(imm8<6>,E-3):imm8<5:4>;
    frac = imm8<3:0>:Zeros(F-4);
    result = sign : exp : frac;

    return result;
```

Library pseudocode for shared/functions/integer/AddWithCarry

```
// AddWithCarry()
// ==============
// Integer addition with carry input, returning result and NZCV flags

(bits(N), bits(4)) AddWithCarry(bits(N) x, bits(N) y, bit carry_in)
    integer unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
    integer signed_sum = SInt(x) + SInt(y) + UInt(carry_in);
    bits(N) result = unsigned_sum<N-1:0>; // same value as signed_sum<N-1:0>
    bit n = result<N-1>;
    bit z = if IsZero(result) then '1' else '0';
    bit c = if UInt(result) == unsigned_sum then '0' else '1';
    bit v = if SInt(result) == signed_sum then '0' else '1';
    return (result, n:z:c:v);
```

Library pseudocode for shared/functions/interrupts/InterruptID

```
enumeration InterruptID {
    InterruptID_PMUIRQ,
    InterruptID_COMMIRQ,
    InterruptID_CTIIRQ,
    InterruptID_COMMRX,
    InterruptID_COMMTX,
    InterruptID_CNTP,
    InterruptID_CNTHP,
    InterruptID_CNTHPS,
    InterruptID_CNTPS,
    InterruptID_CNTV,
    InterruptID_CNTHV,
    InterruptID_CNTHVS,
};
```

Library pseudocode for shared/functions/interrupts/SetInterruptRequestLevel

```
// Set a level-sensitive interrupt to the specified level.
SetInterruptRequestLevel(InterruptID id, signal level);
```

Library pseudocode for shared/functions/memory/AArch64.BranchAddr

```
// AArch64.BranchAddr()
// ====================
// Return the virtual address with tag bits removed.
// This is typically used when the address will be stored to the program counter.

bits(64) AArch64.BranchAddr(bits(64) vaddress, bits(2) el)
    assert !UsingAArch32();
    msbit = AddrTop(vaddress, TRUE, el);
    if msbit == 63 then
        return vaddress;
    elsif (el IN {EL0, EL1} || IsInHost()) && vaddress<msbit> == '1' then
        return SignExtend(vaddress<msbit:0>, 64);
    else
        return ZeroExtend(vaddress<msbit:0>, 64);
```

Library pseudocode for shared/functions/memory/AccType

```
enumeration AccType {AccType_NORMAL,            // Normal loads and stores
                     AccType_STREAM,            // Streaming loads and stores
                     AccType_VEC,               // Vector loads and stores
                     AccType_VECSTREAM,         // Streaming vector loads and stores
                     AccType_SVE,               // Scalable vector loads and stores
                     AccType_SVESTREAM,         // Scalable vector streaming loads and stores
                     AccType_SME,               // Scalable matrix loads and stores
                     AccType_SMESTREAM,         // Scalable matrix streaming loads and stores
                     AccType_UNPRIVSTREAM,      // Streaming unprivileged loads and stores
                     AccType_A32LSMD,           // Load and store multiple
                     AccType_ATOMIC,            // Atomic loads and stores
                     AccType_ATOMICRW,
                     AccType_ORDERED,           // Load-Acquire and Store-Release
                     AccType_ORDEREDRW,
                     AccType_ORDEREDATOMIC,     // Load-Acquire and Store-Release with atomic access
                     AccType_ORDEREDATOMICRW,
                     AccType_ATOMICLS64,        // Atomic 64-byte loads and stores
                     AccType_LIMITEDORDERED,    // Load-LOAcquire and Store-LORelease
                     AccType_UNPRIV,            // Load and store unprivileged
                     AccType_IFETCH,            // Instruction fetch
                     AccType_TTW,               // Translation table walk
                     AccType_NONFAULT,          // Non-faulting loads
                     AccType_CNOTFIRST,         // Contiguous FF load, not first element
                     AccType_NV2REGISTER,       // MRS/MSR instruction used at EL1 and which is
                                                // converted to a memory access that uses the
                                                // EL2 translation regime
                     AccType_TRBE,              // TRBE memory access
                     // Other operations
                     AccType_DC,                // Data cache maintenance
                     AccType_IC,                // Instruction cache maintenance
                     AccType_DCZVA,             // DC ZVA instructions
                     AccType_ATPAN,             // Address translation with PAN permission checks
                     AccType_AT};               // Address translation
```

Library pseudocode for shared/functions/memory/AccessDescriptor

```
type AccessDescriptor is (
    boolean transactional,
    MPAMinfo mpam,
    AccType acctype)
```

Library pseudocode for shared/functions/memory/AddrTop

```
// AddrTop()
// =========
// Return the MSB number of a virtual address in the stage 1 translation regime for "el".
// If EL1 is using AArch64 then addresses from EL0 using AArch32 are zero-extended to 64 bits.

integer AddrTop(bits(64) address, boolean IsInstr, bits(2) el)
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    if ELUsingAArch32(regime) then
        // AArch32 translation regime.
        return 31;
    else
        if EffectiveTBI(address, IsInstr, el) == '1' then
            return 55;
        else
            return 63;
```

Library pseudocode for shared/functions/memory/Allocation

```
constant bits(2) MemHint_No = '00';     // No Read-Allocate, No Write-Allocate
constant bits(2) MemHint_WA = '01';     // No Read-Allocate, Write-Allocate
constant bits(2) MemHint_RA = '10';     // Read-Allocate, No Write-Allocate
constant bits(2) MemHint_RWA = '11';    // Read-Allocate, Write-Allocate
```

Library pseudocode for shared/functions/memory/BigEndian

```
// BigEndian()
// ===========

boolean BigEndian(AccType acctype)
    boolean bigend;
    if HaveNV2Ext() && acctype == AccType_NV2REGISTER then
        return SCTLR_EL2.EE == '1';

    if UsingAArch32() then
        bigend = (PSTATE.E != '0');
    elsif PSTATE.EL == EL0 then
        bigend = (SCTLR[].E0E != '0');
    else
        bigend = (SCTLR[].EE != '0');
    return bigend;
```

Library pseudocode for shared/functions/memory/BigEndianReverse

```
// BigEndianReverse()
// ==================

bits(width) BigEndianReverse (bits(width) value)
    assert width IN {8, 16, 32, 64, 128};
    integer half = width DIV 2;
    if width == 8 then return value;
    return BigEndianReverse(value<half-1:0>) : BigEndianReverse(value<width-1:half>);
```

Library pseudocode for shared/functions/memory/Cacheability

```
constant bits(2) MemAttr_NC = '00';     // Non-cacheable
constant bits(2) MemAttr_WT = '10';     // Write-through
constant bits(2) MemAttr_WB = '11';     // Write-back
```

Library pseudocode for shared/functions/memory/CreateAccessDescriptor

```
// CreateAccessDescriptor()
// ========================

AccessDescriptor CreateAccessDescriptor(AccType acctype)
    AccessDescriptor accdesc;
    accdesc.acctype = acctype;
    accdesc.transactional = FALSE;
    accdesc.mpam = GenMPAMcurEL(acctype);
    return accdesc;
```

Library pseudocode for shared/functions/memory/DataMemoryBarrier

```
DataMemoryBarrier(MBReqDomain domain, MBReqTypes types);
```

Library pseudocode for shared/functions/memory/DataSynchronizationBarrier

```
DataSynchronizationBarrier(MBReqDomain domain, MBReqTypes types, boolean nXS);
```

Library pseudocode for shared/functions/memory/DeviceType

```
enumeration DeviceType {DeviceType_GRE, DeviceType_nGRE, DeviceType_nGnRE, DeviceType_nGnRnE};
```

Library pseudocode for shared/functions/memory/EffectiveTBI

```
// EffectiveTBI()
// ==============
// Returns the effective TBI in the AArch64 stage 1 translation regime for "el".

bit EffectiveTBI(bits(64) address, boolean IsInstr, bits(2) el)
    bit tbi;
    bit tbid;
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));

    case regime of
        when EL1
            tbi = if address<55> == '1' then TCR_EL1.TBI1 else TCR_EL1.TBI0;
            if HavePACExt() then
                tbid = if address<55> == '1' then TCR_EL1.TBID1 else TCR_EL1.TBID0;
        when EL2
            if HaveVirtHostExt() && ELIsInHost(el) then
                tbi = if address<55> == '1' then TCR_EL2.TBI1 else TCR_EL2.TBI0;
                if HavePACExt() then
                    tbid = if address<55> == '1' then TCR_EL2.TBID1 else TCR_EL2.TBID0;
            else
                tbi = TCR_EL2.TBI;
                if HavePACExt() then tbid = TCR_EL2.TBID;
        when EL3
            tbi = TCR_EL3.TBI;
            if HavePACExt() then tbid = TCR_EL3.TBID;

    return (if tbi == '1' && (!HavePACExt() || tbid == '0' || !IsInstr) then '1' else '0');
```

Library pseudocode for shared/functions/memory/EffectiveTCMA

```
// EffectiveTCMA()
// ===============
// Returns the effective TCMA of a virtual address in the stage 1 translation regime for "el".

bit EffectiveTCMA(bits(64) address, bits(2) el)
    bit tcma;
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));

    case regime of
        when EL1
            tcma = if address<55> == '1' then TCR_EL1.TCMA1 else TCR_EL1.TCMA0;
        when EL2
            if HaveVirtHostExt() && ELIsInHost(el) then
                tcma = if address<55> == '1' then TCR_EL2.TCMA1 else TCR_EL2.TCMA0;
            else
                tcma = TCR_EL2.TCMA;
        when EL3
            tcma = TCR_EL3.TCMA;

    return tcma;
```

Library pseudocode for shared/functions/memory/Fault

```
enumeration Fault {Fault_None,
                   Fault_AccessFlag,
                   Fault_Alignment,
                   Fault_Background,
                   Fault_Domain,
                   Fault_Permission,
                   Fault_Translation,
                   Fault_AddressSize,
                   Fault_SyncExternal,
                   Fault_SyncExternalOnWalk,
                   Fault_SyncParity,
                   Fault_SyncParityOnWalk,
                   Fault_GPCFOnWalk,
                   Fault_GPCFOnOutput,
                   Fault_AsyncParity,
                   Fault_AsyncExternal,
                   Fault_Debug,
                   Fault_TLBConflict,
                   Fault_BranchTarget,
                   Fault_HWUpdateAccessFlag,
                   Fault_Lockdown,
                   Fault_Exclusive,
                   Fault_ICacheMaint};
```

Library pseudocode for shared/functions/memory/FaultRecord

```
type FaultRecord is (Fault    statuscode,  // Fault Status
                     AccType  acctype,     // Type of access that faulted
                     FullAddress ipaddress, // Intermediate physical address
                     GPCFRecord gpcf,      // Granule Protection Check Fault record
                     FullAddress paddress,  // Physical address
                     boolean  gpcfs2walk,   // GPC for a stage 2 translation table walk
                     boolean  s2fs1walk,    // Is on a Stage 1 translation table walk
                     boolean  write,        // TRUE for a write, FALSE for a read
                     integer  level,        // For translation, access flag and Permission faults
                     bit      extflag,      // IMPLEMENTATION DEFINED syndrome for External aborts
                     boolean  secondstage,  // Is a Stage 2 abort
                     bits(4)  domain,       // Domain number, AArch32 only
                     bits(2)  errortype,    // [Armv8.2 RAS] AArch32 AET or AArch64 SET
                     bits(4)  debugmoe)     // Debug method of entry, from AArch32 only
```

Library pseudocode for shared/functions/memory/FullAddress

```
type FullAddress is (
    PASpace  paspace,
    bits(52) address
)
```

Library pseudocode for shared/functions/memory/GPCF

```
enumeration GPCF {
    GPCF_None,        // No fault
    GPCF_AddressSize, // GPT address size fault
    GPCF_Walk,        // GPT walk fault
    GPCF_EABT,        // Synchronous External abort on GPT fetch
    GPCF_Fail         // Granule protection fault
};
```

Library pseudocode for shared/functions/memory/GPCFRecord

```
type GPCFRecord is (
    GPCF    gpf,
    integer level
)
```

Library pseudocode for shared/functions/memory/Hint_Prefetch

```
// Signals the memory system that memory accesses of type HINT to or from the specified address are
// likely in the near future. The memory system may take some action to speed up the memory
// accesses when they do occur, such as pre-loading the specified address into one or more
// caches as indicated by the innermost cache level target (0=L1, 1=L2, etc) and non-temporal hint
// stream. Any or all prefetch hints may be treated as a NOP. A prefetch hint must not cause a
// synchronous abort due to Alignment or Translation faults and the like. Its only effect on
// software-visible state should be on caches and TLBs associated with address, which must be
// accessible by reads, writes or execution, as defined in the translation regime of the current
// Exception level. It is guaranteed not to access Device memory.
// A Prefetch_EXEC hint must not result in an access that could not be performed by a speculative
// instruction fetch, therefore if all associated MMUs are disabled, then it cannot access any
// memory location that cannot be accessed by instruction fetches.
Hint_Prefetch(bits(64) address, PrefetchHint hint, integer target, boolean stream);
```

Library pseudocode for shared/functions/memory/IsDataAccess

```
// IsDataAccess()
// ==============
// Return TRUE if access is to data memory.

boolean IsDataAccess(AccType acctype)
    return !(acctype IN {AccType_IFETCH, AccType_TTW,
                         AccType_DC, AccType_IC,
                         AccType_AT, AccType_ATPAN
                        });
```

Library pseudocode for shared/functions/memory/MBReqDomain

```
enumeration MBReqDomain    {MBReqDomain_Nonshareable, MBReqDomain_InnerShareable,
                            MBReqDomain_OuterShareable, MBReqDomain_FullSystem};
```

Library pseudocode for shared/functions/memory/MBReqTypes

```
enumeration MBReqTypes     {MBReqTypes_Reads, MBReqTypes_Writes, MBReqTypes_All};
```

Library pseudocode for shared/functions/memory/MPAM

```
type PARTIDtype = bits(16);
type PMGtype = bits(8);

enumeration PARTIDspaceType {
    PIdSpace_Secure,
    PIdSpace_Root,
    PIdSpace_Realm,
    PIdSpace_NonSecure
};

type MPAMinfo is (
     PARTIDspaceType mpam_sp,
     PARTIDtype partid,
     PMGtype pmg
)
```

Library pseudocode for shared/functions/memory/MemAttrHints

```
type MemAttrHints is (
    bits(2) attrs,  // See MemAttr_*, Cacheability attributes
    bits(2) hints,  // See MemHint_*, Allocation hints
    boolean transient
)
```

Library pseudocode for shared/functions/memory/MemType

```
enumeration MemType {MemType_Normal, MemType_Device};
```

Library pseudocode for shared/functions/memory/MemoryAttributes

```
type MemoryAttributes is (
    MemType      memtype,
    DeviceType   device,       // For Device memory types
    MemAttrHints inner,        // Inner hints and attributes
    MemAttrHints outer,        // Outer hints and attributes
    Shareability shareability, // Shareability attribute
    boolean      tagged,       // Tagged access
    bit          xs            // XS attribute
)
```

Library pseudocode for shared/functions/memory/PASpace

```
enumeration PASpace {
    PAS_NonSecure,
    PAS_Secure,
    PAS_Root,
    PAS_Realm
};
```

Library pseudocode for shared/functions/memory/Permissions

```
type Permissions is (
    bits(2) ap_table,   // Stage 1 hierarchical access permissions
    bit     xn_table,   // Stage 1 hierarchical execute-never for single EL regimes
    bit     pxn_table,  // Stage 1 hierarchical privileged execute-never
    bit     uxn_table,  // Stage 1 hierarchical unprivileged execute-never
    bits(3) ap,         // Stage 1 access permissions
    bit     xn,         // Stage 1 execute-never for single EL regimes
    bit     uxn,        // Stage 1 unprivileged execute-never
    bit     pxn,        // Stage 1 privileged execute-never
    bits(2) s2ap,       // Stage 2 access permissions
    bit     s2xnx,      // Stage 2 extended execute-never
    bit     s2xn        // Stage 2 execute-never
)
```

Library pseudocode for shared/functions/memory/PhysMemRead

```
// Returns the value read from memory, and a status.
// Returned value is UNKNOWN if an External abort occurred while reading the
// memory.
// Otherwise the PhysMemRetStatus statuscode is Fault_None.
(PhysMemRetStatus, bits(8*size)) PhysMemRead(AddressDescriptor desc, integer size,
                                             AccessDescriptor accdesc);
```

Library pseudocode for shared/functions/memory/PhysMemRetStatus

```
type PhysMemRetStatus is (Fault    statuscode,     // Fault Status
                          bit      extflag,        // IMPLEMENTATION DEFINED
                                                   // syndrome for External aborts
                          bits(2)  errortype,      // optional error state
                                                   // returned on a physical
                                                   // memory access
                          bits(64) store64bstatus, // status of 64B store
                          AccType  acctype)        // Type of access that faulted
```

Library pseudocode for shared/functions/memory/PhysMemWrite

```
// Writes the value to memory, and returns the status of the write.
// If there is an External abort on the write, the PhysMemRetStatus indicates this.
// Otherwise the statuscode of PhysMemRetStatus is Fault_None.
PhysMemRetStatus PhysMemWrite(AddressDescriptor desc, integer size, AccessDescriptor accdesc,
                              bits(8*size) value);
```

Library pseudocode for shared/functions/memory/PrefetchHint

```
enumeration PrefetchHint {Prefetch_READ, Prefetch_WRITE, Prefetch_EXEC};
```

Library pseudocode for shared/functions/memory/Shareability

```
enumeration Shareability {
    Shareability_NSH,
    Shareability_ISH,
    Shareability_OSH
};
```

Library pseudocode for shared/functions/memory/SpeculativeStoreBypassBarrierToPA

```
SpeculativeStoreBypassBarrierToPA();
```

Library pseudocode for shared/functions/memory/SpeculativeStoreBypassBarrierToVA

```
SpeculativeStoreBypassBarrierToVA();
```

Library pseudocode for shared/functions/memory/Tag

```
constant integer LOG2_TAG_GRANULE = 4;

constant integer TAG_GRANULE = 1 << LOG2_TAG_GRANULE;
```

Library pseudocode for shared/functions/mpam/AltPARTIDspace

```
// AltPARTIDspace()
// ================
// From the Security state, EL and ALTSP configuration, determine
// whether to primary space or the alt space is selected and which
// PARTID space is the alternative space. Return that alternative
// PARTID space if selected or the primary space if not.

PARTIDspaceType AltPARTIDspace(bits(2) el, SecurityState security,
                               PARTIDspaceType primaryPIdSpace)
    case security of
        when SS_NonSecure
            assert el != EL3;
            return primaryPIdSpace; // there is no ALTSP for Non_secure
        when SS_Secure
            assert el != EL3;
            if primaryPIdSpace == PIdSpace_NonSecure then
                return primaryPIdSpace;
            return AltPIdSecure(el, primaryPIdSpace);
        when SS_Root
            assert el == EL3;
            if MPAM3_EL3.ALTSP_EL3 == '1' then
                if MPAM3_EL3.RT_ALTSP_NS == '1' then
                    return PIdSpace_NonSecure;
                else
                    return PIdSpace_Secure;
            else
                return primaryPIdSpace;
        when SS_Realm
            assert el != EL3;
            return AltPIdRealm(el, primaryPIdSpace);
        otherwise
            Unreachable();
```

Library pseudocode for shared/functions/mpam/AltPIdRealm

```
// AltPIdRealm()
// =============
// Compute PARTID space as either the primary PARTID space or
// alternative PARTID space in the Realm Security state.
// Helper for AltPARTIDspace.

PARTIDspaceType AltPIdRealm(bits(2) el, PARTIDspaceType primaryPIdSpace)
    PARTIDspaceType PIdSpace = primaryPIdSpace;
    case el of
        when EL0
            if ELIsInHost(EL0) then
                if !UsePrimarySpaceEL2() then
                    PIdSpace = PIdSpace_NonSecure;
            elsif !UsePrimarySpaceEL10() then
                PIdSpace = PIdSpace_NonSecure;
        when EL1
            if !UsePrimarySpaceEL10() then
                PIdSpace = PIdSpace_NonSecure;
        when EL2
            if !UsePrimarySpaceEL2() then
                PIdSpace = PIdSpace_NonSecure;
        otherwise
            Unreachable();
    return PIdSpace;
```

Library pseudocode for shared/functions/mpam/AltPIdSecure

```
// AltPIdSecure()
// ==============
// Compute PARTID space as either the primary PARTID space or
// alternative PARTID space in the Secure Security state.
// Helper for AltPARTIDspace.

PARTIDspaceType AltPIdSecure(bits(2) el, PARTIDspaceType primaryPIdSpace)
    PARTIDspaceType PIdSpace = primaryPIdSpace;
    boolean el2en = EL2Enabled();
    case el of
        when EL0
            if el2en then
                if ELIsInHost(EL0) then
                    if !UsePrimarySpaceEL2() then
                        PIdSpace = PIdSpace_NonSecure;
                elsif !UsePrimarySpaceEL10() then
                    PIdSpace = PIdSpace_NonSecure;
            elsif MPAM3_EL3.ALTSP_HEN == '0' && MPAM3_EL3.ALTSP_HFC == '1' then
                PIdSpace = PIdSpace_NonSecure;
        when EL1
            if el2en then
                if !UsePrimarySpaceEL10() then
                    PIdSpace = PIdSpace_NonSecure;
            elsif MPAM3_EL3.ALTSP_HEN == '0' && MPAM3_EL3.ALTSP_HFC == '1' then
                PIdSpace = PIdSpace_NonSecure;
        when EL2
            if !UsePrimarySpaceEL2() then
                PIdSpace = PIdSpace_NonSecure;
        otherwise
            Unreachable();
    return PIdSpace;
```

Library pseudocode for shared/functions/mpam/DefaultMPAMinfo

```
// DefaultMPAMinfo()
// =================
// Returns default MPAM info.  The partidspace argument sets
// the PARTID space of the default MPAM information returned.

MPAMinfo DefaultMPAMinfo(PARTIDspaceType partidspace)
    MPAMinfo DefaultInfo;
    DefaultInfo.mpam_sp = partidspace;
    DefaultInfo.partid  = DefaultPARTID;
    DefaultInfo.pmg     = DefaultPMG;
    return DefaultInfo;
```

Library pseudocode for shared/functions/mpam/DefaultPARTID

```
constant PARTIDtype DefaultPARTID = 0<15:0>;
```

Library pseudocode for shared/functions/mpam/DefaultPMG

```
constant PMGtype    DefaultPMG = 0<7:0>;
```

Library pseudocode for shared/functions/mpam/GenMPAMcurEL

```
// GenMPAMcurEL()
// ==============
// Returns MPAMinfo for the current EL and security state.
// May be called if MPAM is not implemented (but in an version that supports
// MPAM), MPAM is disabled, or in AArch32.  In AArch32, convert the mode to
// EL if can and use that to drive MPAM information generation.  If mode
// cannot be converted, MPAM is not implemented, or MPAM is disabled return
// default MPAM information for the current security state.

MPAMinfo GenMPAMcurEL(AccType acctype)
    bits(2) mpamEL;
    boolean validEL = FALSE;
    SecurityState security = CurrentSecurityState();
    boolean InD = FALSE;
    boolean InSM = FALSE;
    PARTIDspaceType pspace = PARTIDspaceFromSS(security);
    if pspace == PIdSpace_NonSecure && !MPAMisEnabled() then
        return DefaultMPAMinfo(pspace);
    if UsingAArch32() then
        (validEL, mpamEL) = ELFromM32(PSTATE.M);
    else
        mpamEL = if acctype == AccType_NV2REGISTER then EL2 else PSTATE.EL;
        validEL = TRUE;
    case acctype of
        when AccType_IFETCH, AccType_IC
            InD = TRUE;
        when AccType_SME, AccType_SMESTREAM
            InSM = (boolean IMPLEMENTATION_DEFINED "Shared SMCU" ||
                    boolean IMPLEMENTATION_DEFINED "MPAMSM_EL1 label precedence");
        when AccType_VEC, AccType_VECSTREAM
            InSM = (HaveSME() && PSTATE.SM == '1' &&
                    (boolean IMPLEMENTATION_DEFINED "Shared SMCU" ||
                    boolean IMPLEMENTATION_DEFINED "MPAMSM_EL1 label precedence"));
        when AccType_SVE, AccType_SVESTREAM, AccType_NONFAULT, AccType_CNOTFIRST
            InSM = (HaveSME() && PSTATE.SM == '1' &&
                    (boolean IMPLEMENTATION_DEFINED "Shared SMCU" ||
                    boolean IMPLEMENTATION_DEFINED "MPAMSM_EL1 label precedence"));
        otherwise
            // Other access types are DATA accesses
            InD = FALSE;
    if !validEL then
        return DefaultMPAMinfo(pspace);
    elsif HaveRME() && MPAMIDR_EL1.HAS_ALTSP == '1' then
        // Substitute alternative PARTID space if selected
        pspace = AltPARTIDspace(mpamEL, security, pspace);
    if HaveMPAMv0p1Ext() && MPAMIDR_EL1.HAS_FORCE_NS == '1' then
        if MPAM3_EL3.FORCE_NS == '1' && security == SS_Secure then
            pspace = PIdSpace_NonSecure;
    if (HaveMPAMv0p1Ext() || HaveMPAMv1p1Ext()) && MPAMIDR_EL1.HAS_SDEFLT == '1' then
        if MPAM3_EL3.SDEFLT == '1' && security == SS_Secure then
            return DefaultMPAMinfo(pspace);
    if !MPAMisEnabled() then
        return DefaultMPAMinfo(pspace);
    else
        return genMPAM(mpamEL, InD, InSM, pspace);
```

Library pseudocode for shared/functions/mpam/MAP_vPARTID

```
// MAP_vPARTID()
// =============
// Performs conversion of virtual PARTID into physical PARTID
// Contains all of the error checking and implementation
// choices for the conversion.

(PARTIDtype, boolean) MAP_vPARTID(PARTIDtype vpartid)
    // should not ever be called if EL2 is not implemented
    // or is implemented but not enabled in the current
    // security state.
    PARTIDtype ret;
    boolean err;
    integer virt    = UInt(vpartid);
    integer vpmrmax = UInt(MPAMIDR_EL1.VPMR_MAX);

    // vpartid_max is largest vpartid supported
    integer vpartid_max = (vpmrmax << 2) + 3;

    // One of many ways to reduce vpartid to value less than vpartid_max.
    if UInt(vpartid) > vpartid_max then
        virt = virt MOD (vpartid_max+1);

    // Check for valid mapping entry.
    if MPAMVPMV_EL2<virt> == '1' then
        // vpartid has a valid mapping so access the map.
        ret = mapvpmw(virt);
        err = FALSE;

    // Is the default virtual PARTID valid?
    elsif MPAMVPMV_EL2<0> == '1' then
        // Yes, so use default mapping for vpartid == 0.
        ret = MPAMVPM0_EL2<0 +: 16>;
        err = FALSE;

    // Neither is valid so use default physical PARTID.
    else
        ret = DefaultPARTID;
        err = TRUE;

    // Check that the physical PARTID is in-range.
    // This physical PARTID came from a virtual mapping entry.
    integer partid_max = UInt(MPAMIDR_EL1.PARTID_MAX);
    if UInt(ret) > partid_max then
        // Out of range, so return default physical PARTID
        ret = DefaultPARTID;
        err = TRUE;
    return (ret, err);
```

Library pseudocode for shared/functions/mpam/MPAMisEnabled

```
// MPAMisEnabled()
// ===============
// Returns TRUE if MPAMisEnabled.

boolean MPAMisEnabled()
    el = HighestEL();
    case el of
        when EL3 return MPAM3_EL3.MPAMEN == '1';
        when EL2 return MPAM2_EL2.MPAMEN == '1';
        when EL1 return MPAM1_EL1.MPAMEN == '1';
```

Library pseudocode for shared/functions/mpam/MPAMisVirtual

```
// MPAMisVirtual()
// ===============
// Returns TRUE if MPAM is configured to be virtual at EL.

boolean MPAMisVirtual(bits(2) el)
    return (MPAMIDR_EL1.HAS_HCR == '1' && EL2Enabled() &&
            ((el == EL0 && MPAMHCR_EL2.EL0_VPMEN == '1' &&
               (HCR_EL2.E2H == '0' || HCR_EL2.TGE == '0')) ||
             (el == EL1 && MPAMHCR_EL2.EL1_VPMEN == '1')));
```

Library pseudocode for shared/functions/mpam/PARTIDspaceFromSS

```
// PARTIDspaceFromSS()
// ===================
// Returns the primary PARTID space from the Security State.

PARTIDspaceType PARTIDspaceFromSS(SecurityState security)
    case security of
        when SS_NonSecure
            return PIdSpace_NonSecure;
        when SS_Root
            return PIdSpace_Root;
        when SS_Realm
            return PIdSpace_Realm;
        when SS_Secure
            return PIdSpace_Secure;
        otherwise
            Unreachable();
```

Library pseudocode for shared/functions/mpam/UsePrimarySpaceEL10

```
// UsePrimarySpaceEL10()
// =====================
// Checks whether Primary space is configured in the
// MPAM3_EL3 and MPAM2_EL2 ALTSP control bits that affect
// MPAM ALTSP use at EL1 and EL0.

boolean UsePrimarySpaceEL10()
    if MPAM3_EL3.ALTSP_HEN == '0' then
        return MPAM3_EL3.ALTSP_HFC == '0';
    return !MPAMisEnabled() || !EL2Enabled() || MPAM2_EL2.ALTSP_HFC == '0';
```

Library pseudocode for shared/functions/mpam/UsePrimarySpaceEL2

```
// UsePrimarySpaceEL2()
// ====================
// Checks whether Primary space is configured in the
// MPAM3_EL3 and MPAM2_EL2 ALTSP control bits that affect
// MPAM ALTSP use at EL2.

boolean UsePrimarySpaceEL2()
    if MPAM3_EL3.ALTSP_HEN == '0' then
        return MPAM3_EL3.ALTSP_HFC == '0';
    return !MPAMisEnabled() || MPAM2_EL2.ALTSP_EL2 == '0';
```

Library pseudocode for shared/functions/mpam/genMPAM

```
// genMPAM()
// =========
// Returns MPAMinfo for exception level el.
// If InD is TRUE returns MPAM information using PARTID_I and PMG_I fields
// of MPAMel_ELx register and otherwise using PARTID_D and PMG_D fields.
// If InSM is TRUE returns MPAM information using PARTID_D and PMG_D fields
// of MPAMSM_EL1 register.
// Produces a PARTID in PARTID space pspace.

MPAMinfo genMPAM(bits(2) el, boolean InD, boolean InSM, PARTIDspaceType pspace)
    MPAMinfo returninfo;
    PARTIDtype partidel;
    boolean perr;
    // gstplk is guest OS application locked by the EL2 hypervisor to
    // only use EL1 the virtual machine's PARTIDs.
    boolean gstplk = (el == EL0 && EL2Enabled() &&
                      MPAMHCR_EL2.GSTAPP_PLK == '1' &&
                      HCR_EL2.TGE == '0');
    bits(2) eff_el = if gstplk then EL1 else el;
    (partidel, perr) = genPARTID(eff_el, InD, InSM);
    PMGtype groupel  = genPMG(eff_el, InD, InSM, perr);
    returninfo.mpam_sp = pspace;
    returninfo.partid  = partidel;
    returninfo.pmg     = groupel;
    return returninfo;
```

Library pseudocode for shared/functions/mpam/genPARTID

```
// genPARTID()
// ===========
// Returns physical PARTID and error boolean for exception level el.
// If InD is TRUE then PARTID is from MPAMel_ELx.PARTID_I and
// otherwise from MPAMel_ELx.PARTID_D.
// If InSM is TRUE then PARTID is from MPAMSM_EL1.PARTID_D.

(PARTIDtype, boolean) genPARTID(bits(2) el, boolean InD, boolean InSM)
    PARTIDtype partidel = getMPAM_PARTID(el, InD, InSM);
    PARTIDtype partid_max = MPAMIDR_EL1.PARTID_MAX;
    if UInt(partidel) > UInt(partid_max) then
        return (DefaultPARTID, TRUE);
    if MPAMisVirtual(el) then
        return MAP_vPARTID(partidel);
    else
        return (partidel, FALSE);
```

Library pseudocode for shared/functions/mpam/genPMG

```
// genPMG()
// ========
// Returns PMG for exception level el and I- or D-side (InD).
// If PARTID generation (genPARTID) encountered an error, genPMG() should be
// called with partid_err as TRUE.

PMGtype genPMG(bits(2) el, boolean InD, boolean InSM, boolean partid_err)
    integer pmg_max = UInt(MPAMIDR_EL1.PMG_MAX);
    // It is CONSTRAINED UNPREDICTABLE whether partid_err forces PMG to
    // use the default or if it uses the PMG from getMPAM_PMG.
    if partid_err then
        return DefaultPMG;
    PMGtype groupel = getMPAM_PMG(el, InD, InSM);
    if UInt(groupel) <= pmg_max then
        return groupel;
    return DefaultPMG;
```

Library pseudocode for shared/functions/mpam/getMPAM_PARTID

```
// getMPAM_PARTID()
// ================
// Returns a PARTID from one of the MPAMn_ELx or MPAMSM_EL1 registers.
// If InSM is TRUE, the MPAMSM_EL1 register is used. Otherwise,
// MPAMn selects the MPAMn_ELx register used.
// If InD is TRUE, selects the PARTID_I field of that
// register.  Otherwise, selects the PARTID_D field.

PARTIDtype getMPAM_PARTID(bits(2) MPAMn, boolean InD, boolean InSM)
    PARTIDtype partid;
    boolean el2avail = EL2Enabled();

    if InSM then
        partid = MPAMSM_EL1.PARTID_D;
        return partid;

    if InD then
        case MPAMn of
            when '11' partid = MPAM3_EL3.PARTID_I;
            when '10' partid = if el2avail then MPAM2_EL2.PARTID_I else Zeros(16);
            when '01' partid = MPAM1_EL1.PARTID_I;
            when '00' partid = MPAM0_EL1.PARTID_I;
            otherwise partid = PARTIDtype UNKNOWN;
    else
        case MPAMn of
            when '11' partid = MPAM3_EL3.PARTID_D;
            when '10' partid = if el2avail then MPAM2_EL2.PARTID_D else Zeros(16);
            when '01' partid = MPAM1_EL1.PARTID_D;
            when '00' partid = MPAM0_EL1.PARTID_D;
            otherwise partid = PARTIDtype UNKNOWN;
    return partid;
```

Library pseudocode for shared/functions/mpam/getMPAM_PMG

```
// getMPAM_PMG()
// =============
// Returns a PMG from one of the MPAMn_ELx or MPAMSM_EL1 registers.
// If InSM is TRUE, the MPAMSM_EL1 register is used. Otherwise,
// MPAMn selects the MPAMn_ELx register used.
// If InD is TRUE, selects the PMG_I field of that
// register.  Otherwise, selects the PMG_D field.

PMGtype getMPAM_PMG(bits(2) MPAMn, boolean InD, boolean InSM)
    PMGtype pmg;
    boolean el2avail = EL2Enabled();

    if InSM then
        pmg = MPAMSM_EL1.PMG_D;
        return pmg;

    if InD then
        case MPAMn of
            when '11' pmg = MPAM3_EL3.PMG_I;
            when '10' pmg = if el2avail then MPAM2_EL2.PMG_I else Zeros(8);
            when '01' pmg = MPAM1_EL1.PMG_I;
            when '00' pmg = MPAM0_EL1.PMG_I;
            otherwise pmg = PMGtype UNKNOWN;
    else
        case MPAMn of
            when '11' pmg = MPAM3_EL3.PMG_D;
            when '10' pmg = if el2avail then MPAM2_EL2.PMG_D else Zeros(8);
            when '01' pmg = MPAM1_EL1.PMG_D;
            when '00' pmg = MPAM0_EL1.PMG_D;
            otherwise pmg = PMGtype UNKNOWN;
    return pmg;
```

Library pseudocode for shared/functions/mpam/mapvpmw

```
// mapvpmw()
// =========
// Map a virtual PARTID into a physical PARTID using
// the MPAMVPMn_EL2 registers.
// vpartid is now assumed in-range and valid (checked by caller)
// returns physical PARTID from mapping entry.

PARTIDtype mapvpmw(integer vpartid)
    bits(64) vpmw;
    integer  wd = vpartid DIV 4;
    case wd of
        when 0 vpmw = MPAMVPM0_EL2;
        when 1 vpmw = MPAMVPM1_EL2;
        when 2 vpmw = MPAMVPM2_EL2;
        when 3 vpmw = MPAMVPM3_EL2;
        when 4 vpmw = MPAMVPM4_EL2;
        when 5 vpmw = MPAMVPM5_EL2;
        when 6 vpmw = MPAMVPM6_EL2;
        when 7 vpmw = MPAMVPM7_EL2;
        otherwise vpmw = Zeros(64);
    // vpme_lsb selects LSB of field within register
    integer vpme_lsb = (vpartid MOD 4) * 16;
    return vpmw<vpme_lsb +: 16>;
```

Library pseudocode for shared/functions/predictionrestrict/ASID

```
// ASID[]
// ======
// Effective ASID.

bits(16) ASID[]
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H, TGE> == '11' then
        if TCR_EL2.A1 == '1' then
            return TTBR1_EL2.ASID;
        else
            return TTBR0_EL2.ASID;

    elsif !ELUsingAArch32(EL1) then
        if TCR_EL1.A1 == '1' then
            return TTBR1_EL1.ASID;
        else
            return TTBR0_EL1.ASID;

    else
        if TTBCR.EAE == '0' then
            return ZeroExtend(CONTEXTIDR.ASID, 16);
        else
            if TTBCR.A1 == '1' then
                return ZeroExtend(TTBR1.ASID, 16);
            else
                return ZeroExtend(TTBR0.ASID, 16);
```

Library pseudocode for shared/functions/predictionrestrict/ExecutionCntxt

```
type ExecutionCntxt is (
    boolean          is_vmid_valid, // is vmid valid for current context
    boolean          all_vmid,      // should the operation be applied for all vmids
    bits(16)         vmid,          // if all_vmid = FALSE, vmid to which operation is applied
    boolean          is_asid_valid, // is asid valid for current context
    boolean          all_asid,      // should the operation be applied for all asids
    bits(16)         asid,          // if all_asid = FALSE, ASID to which operation is applied
    bits(2)          target_el,     // target EL at which operation is performed
    SecurityState    security,
    RestrictType     restriction    // type of restriction operation
)
```

Library pseudocode for shared/functions/predictionrestrict/RESTRICT_PREDICTIONS

```
// RESTRICT_PREDICTIONS()
// ======================
// Clear all speculated values.

RESTRICT_PREDICTIONS(ExecutionCntxt c)
    IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/functions/predictionrestrict/RestrictType

```
enumeration RestrictType {
    RestrictType_DataValue,
    RestrictType_ControlFlow,
    RestrictType_CachePrefetch
};
```

Library pseudocode for shared/functions/predictionrestrict/TargetSecurityState

```
// TargetSecurityState()
// =====================
// Decode the target security state for the prediction context.

SecurityState TargetSecurityState(bit NS, bit NSE)
    curr_ss =  SecurityStateAtEL(PSTATE.EL);
    if curr_ss == SS_NonSecure then
        return SS_NonSecure;
    elsif curr_ss == SS_Secure then
        case NS of
            when '0' return SS_Secure;
            when '1' return SS_NonSecure;
    elsif HaveRME() then
        if curr_ss == SS_Root then
            case NSE:NS of
                when '00' return SS_Secure;
                when '01' return SS_NonSecure;
                when '11' return SS_Realm;
                when '10' return SS_Root;
        elsif curr_ss == SS_Realm then
            return SS_Realm;
```

Library pseudocode for shared/functions/registers/BranchTo

```
// BranchTo()
// ==========
// Set program counter to a new address, with a branch type.
// Parameter branch_conditional indicates whether the executed branch has a conditional encoding.
// In AArch64 state the address might include a tag in the top eight bits.

BranchTo(bits(N) target, BranchType branch_type, boolean branch_conditional)
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target, 64);
    else
        assert N == 64 && !UsingAArch32();
        bits(64) target_vaddress = AArch64.BranchAddr(target<63:0>, PSTATE.EL);
        if (HaveBRBExt() &&
            branch_type IN {BranchType_DIR, BranchType_INDIR,
                            BranchType_DIRCALL, BranchType_INDCALL,
                            BranchType_RET}) then
            BRBEBranch(branch_type, branch_conditional, target_vaddress);

        _PC = target_vaddress;
    return;
```

Library pseudocode for shared/functions/registers/BranchToAddr

```
// BranchToAddr()
// ==============
// Set program counter to a new address, with a branch type.
// In AArch64 state the address does not include a tag in the top eight bits.

BranchToAddr(bits(N) target, BranchType branch_type)
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target, 64);
    else
        assert N == 64 && !UsingAArch32();
        _PC = target<63:0>;
    return;
```

Library pseudocode for shared/functions/registers/BranchType

```
enumeration BranchType {
    BranchType_DIRCALL,     // Direct Branch with link
    BranchType_INDCALL,     // Indirect Branch with link
    BranchType_ERET,        // Exception return (indirect)
    BranchType_DBGEXIT,     // Exit from Debug state
    BranchType_RET,         // Indirect branch with function return hint
    BranchType_DIR,         // Direct branch
    BranchType_INDIR,       // Indirect branch
    BranchType_EXCEPTION,   // Exception entry
    BranchType_TMFAIL,      // Transaction failure
    BranchType_RESET,       // Reset
    BranchType_UNKNOWN};    // Other
```

Library pseudocode for shared/functions/registers/Hint_Branch

```
// Report the hint passed to BranchTo() and BranchToAddr(), for consideration when processing
// the next instruction.
Hint_Branch(BranchType hint);
```

Library pseudocode for shared/functions/registers/NextInstrAddr

```
// Return address of the sequentially next instruction.
bits(N) NextInstrAddr(integer N);
```

Library pseudocode for shared/functions/registers/ResetExternalDebugRegisters

```
// Reset the External Debug registers in the Core power domain.
ResetExternalDebugRegisters(boolean cold_reset);
```

Library pseudocode for shared/functions/registers/ThisInstrAddr

```
// ThisInstrAddr()
// ===============
// Return address of the current instruction.

bits(N) ThisInstrAddr(integer N)
    assert N == 64 || (N == 32 && UsingAArch32());
    return _PC<N-1:0>;
```

Library pseudocode for shared/functions/registers/_PC

```
bits(64) _PC;
```

Library pseudocode for shared/functions/registers/_R

```
array bits(64) _R[0..30];
```

Library pseudocode for shared/functions/sysregisters/SPSR

```
// SPSR[] - non-assignment form
// ============================

bits(N) SPSR[]
    bits(N) result;
    if UsingAArch32() then
        assert N == 32;
        case PSTATE.M of
            when M32_FIQ      result = SPSR_fiq<N-1:0>;
            when M32_IRQ      result = SPSR_irq<N-1:0>;
            when M32_Svc      result = SPSR_svc<N-1:0>;
            when M32_Monitor  result = SPSR_mon<N-1:0>;
            when M32_Abort    result = SPSR_abt<N-1:0>;
            when M32_Hyp      result = SPSR_hyp<N-1:0>;
            when M32_Undef    result = SPSR_und<N-1:0>;
            otherwise         Unreachable();
    else
        assert N == 64;
        case PSTATE.EL of
            when EL1          result = SPSR_EL1<N-1:0>;
            when EL2          result = SPSR_EL2<N-1:0>;
            when EL3          result = SPSR_EL3<N-1:0>;
            otherwise         Unreachable();
    return result;

// SPSR[] - assignment form
// ========================

SPSR[] = bits(N) value
    if UsingAArch32() then
        assert N == 32;
        case PSTATE.M of
            when M32_FIQ      SPSR_fiq<N-1:0> = value<N-1:0>;
            when M32_IRQ      SPSR_irq<N-1:0> = value<N-1:0>;
            when M32_Svc      SPSR_svc<N-1:0> = value<N-1:0>;
            when M32_Monitor  SPSR_mon<N-1:0> = value<N-1:0>;
            when M32_Abort    SPSR_abt<N-1:0> = value<N-1:0>;
            when M32_Hyp      SPSR_hyp<N-1:0> = value<N-1:0>;
            when M32_Undef    SPSR_und<N-1:0> = value<N-1:0>;
            otherwise         Unreachable();
    else
        assert N == 64;
        case PSTATE.EL of
            when EL1          SPSR_EL1<N-1:0> = value<N-1:0>;
            when EL2          SPSR_EL2<N-1:0> = value<N-1:0>;
            when EL3          SPSR_EL3<N-1:0> = value<N-1:0>;
            otherwise         Unreachable();
    return;
```

Library pseudocode for shared/functions/system/ArchVersion

```
enumeration ArchVersion {
    ARMv8p0
    , ARMv8p1
    , ARMv8p2
    , ARMv8p3
    , ARMv8p4
    , ARMv8p5
    , ARMv8p6
    , ARMv8p7
    , ARMv8p8
};
```

Library pseudocode for shared/functions/system/BranchTargetCheck

```
// BranchTargetCheck()
// ===================
// This function is executed checks if the current instruction is a valid target for a branch
// taken into, or inside, a guarded page. It is executed on every cycle once the current
// instruction has been decoded and the values of InGuardedPage and BTypeCompatible have been
// determined for the current instruction.

BranchTargetCheck()
    assert HaveBTIExt() && !UsingAArch32();

    // The branch target check considers two state variables:
    // * InGuardedPage, which is evaluated during instruction fetch.
    // * BTypeCompatible, which is evaluated during instruction decode.
    if InGuardedPage && PSTATE.BTYPE != '00' && !BTypeCompatible && !Halted() then
        bits(64) pc = ThisInstrAddr(64);
        AArch64.BranchTargetException(pc<51:0>);

    boolean branch_instr = AArch64.ExecutingBROrBLROrRetInstr();
    boolean bti_instr    = AArch64.ExecutingBTIInstr();

    // PSTATE.BTYPE defaults to 00 for instructions that do not explictly set BTYPE.
    if !(branch_instr || bti_instr) then
        BTypeNext = '00';
```

Library pseudocode for shared/functions/system/ClearEventRegister

```
// ClearEventRegister()
// ====================
// Clear the Event Register of this PE.

ClearEventRegister()
    EventRegister = '0';
    return;
```

Library pseudocode for shared/functions/system/ClearPendingPhysicalSError

```
// Clear a pending physical SError interrupt.
ClearPendingPhysicalSError();
```

Library pseudocode for shared/functions/system/ClearPendingVirtualSError

```
// Clear a pending virtual SError interrupt.
ClearPendingVirtualSError();
```

Library pseudocode for shared/functions/system/ConditionHolds

```
// ConditionHolds()
// ================
// Return TRUE iff COND currently holds

boolean ConditionHolds(bits(4) cond)
    // Evaluate base condition.
    boolean result;
    case cond<3:1> of
        when '000' result = (PSTATE.Z == '1');                          // EQ or NE
        when '001' result = (PSTATE.C == '1');                          // CS or CC
        when '010' result = (PSTATE.N == '1');                          // MI or PL
        when '011' result = (PSTATE.V == '1');                          // VS or VC
        when '100' result = (PSTATE.C == '1' && PSTATE.Z == '0');       // HI or LS
        when '101' result = (PSTATE.N == PSTATE.V);                     // GE or LT
        when '110' result = (PSTATE.N == PSTATE.V && PSTATE.Z == '0');  // GT or LE
        when '111' result = TRUE;                                       // AL

    // Condition flag values in the set '111x' indicate always true
    // Otherwise, invert condition if necessary.
    if cond<0> == '1' && cond != '1111' then
        result = !result;

    return result;
```

Library pseudocode for shared/functions/system/ConsumptionOfSpeculativeDataBarrier

```
ConsumptionOfSpeculativeDataBarrier();
```

Library pseudocode for shared/functions/system/CurrentInstrSet

```
// CurrentInstrSet()
// =================

InstrSet CurrentInstrSet()
    InstrSet result;
    if UsingAArch32() then
        result = if PSTATE.T == '0' then InstrSet_A32 else InstrSet_T32;
        // PSTATE.J is RES0. Implementation of T32EE or Jazelle state not permitted.
    else
        result = InstrSet_A64;
    return result;
```

Library pseudocode for shared/functions/system/CurrentPL

```
// CurrentPL()
// ===========

PrivilegeLevel CurrentPL()
    return PLOfEL(PSTATE.EL);
```

Library pseudocode for shared/functions/system/CurrentSecurityState

```
// CurrentSecurityState()
// ======================
// Returns the effective security state at the exception level based off current settings.

SecurityState CurrentSecurityState()
    return SecurityStateAtEL(PSTATE.EL);
```

Library pseudocode for shared/functions/system/DSBAlias

```
enumeration DSBAlias {DSBAlias_SSBB, DSBAlias_PSSBB, DSBAlias_DSB};
```

Library pseudocode for shared/functions/system/EL0

```
constant bits(2) EL3 = '11';
constant bits(2) EL2 = '10';
constant bits(2) EL1 = '01';
constant bits(2) EL0 = '00';
```

Library pseudocode for shared/functions/system/EL2Enabled

```
// EL2Enabled()
// ============
// Returns TRUE if EL2 is present and executing
// - with the PE in Non-secure state when Non-secure EL2 is implemented, or
// - with the PE in Realm state when Realm EL2 is implemented, or
// - with the PE in Secure state when Secure EL2 is implemented and enabled, or
// - when EL3 is not implemented.

boolean EL2Enabled()
    return HaveEL(EL2) && (!HaveEL(EL3) || SCR_GEN[].NS == '1' || IsSecureEL2Enabled());
```

Library pseudocode for shared/functions/system/EL3SDDTrapPriority

```
// EL3SDDTrapPriority()
// ====================
// Returns TRUE if in Debug state, EDSCR.SDD is set, and an EL3 trap by an
// EL3 control register has the priority of the original trap exception.
// The IMPLEMENTATION DEFINED priority may be different for each case.

boolean EL3SDDTrapPriority()
    return (Halted() && EDSCR.SDD == '1' &&
            boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'");
```

Library pseudocode for shared/functions/system/ELFromM32

```
// ELFromM32()
// ===========

(boolean,bits(2)) ELFromM32(bits(5) mode)
    // Convert an AArch32 mode encoding to an Exception level.
    // Returns (valid,EL):
    //   'valid' is TRUE if 'mode<4:0>' encodes a mode that is both valid for this implementation
    //           and the current value of SCR.NS/SCR_EL3.NS.
    //   'EL'    is the Exception level decoded from 'mode'.
    bits(2) el;
    boolean valid = !BadMode(mode);  // Check for modes that are not valid for this implementation
    case mode of
        when M32_Monitor
            el = EL3;
        when M32_Hyp
            el = EL2;
        when M32_FIQ, M32_IRQ, M32_Svc, M32_Abort, M32_Undef, M32_System
            // If EL3 is implemented and using AArch32, then these modes are EL3 modes in Secure
            // state, and EL1 modes in Non-secure state. If EL3 is not implemented or is using
            // AArch64, then these modes are EL1 modes.
            el = (if HaveEL(EL3) && !HaveAArch64() && SCR.NS == '0' then EL3 else EL1);
        when M32_User
            el = EL0;
        otherwise
            valid = FALSE;           // Passed an illegal mode value

    if valid && el == EL2 && HaveEL(EL3) && SCR_GEN[].NS == '0' then
        valid = FALSE;               // EL2 only valid in Non-secure state in AArch32
    elsif valid && HaveRME() && SCR_EL3.<NSE,NS> == '10' then
        valid = FALSE;               // No AArch32 modes in Root state

    if !valid then el = bits(2) UNKNOWN;
    return (valid, el);
```

Library pseudocode for shared/functions/system/ELFromSPSR

```
// ELFromSPSR()
// ============

// Convert an SPSR value encoding to an Exception level.
// Returns (valid,EL):
//   'valid' is TRUE if 'spsr<4:0>' encodes a valid mode for the current state.
//   'EL'    is the Exception level decoded from 'spsr'.

(boolean,bits(2)) ELFromSPSR(bits(N) spsr)
    bits(2) el;
    boolean valid;
    if spsr<4> == '0' then                      // AArch64 state
        el = spsr<3:2>;
        if !HaveAArch64() then                  // No AArch64 support
            valid = FALSE;
        elsif !HaveEL(el) then                  // Exception level not implemented
            valid = FALSE;
        elsif spsr<1> == '1' then               // M[1] must be 0
            valid = FALSE;
        elsif el == EL0 && spsr<0> == '1' then  // for EL0, M[0] must be 0
            valid = FALSE;
        elsif HaveRME() && el != EL3 && SCR_EL3.<NSE,NS> == '10' then
            valid = FALSE;                      // Only EL3 valid in Root state
        elsif el == EL2 && HaveEL(EL3) && !IsSecureEL2Enabled() && SCR_EL3.NS == '0' then
            valid = FALSE;                      // Unless Secure EL2 is enabled, EL2 valid only in Non-se
        else
            valid = TRUE;
    elsif HaveAArch32() then                    // AArch32 state
        (valid, el) = ELFromM32(spsr<4:0>);
    else
        valid = FALSE;

    if !valid then el = bits(2) UNKNOWN;
    return (valid,el);
```

Library pseudocode for shared/functions/system/ELIsInHost

```
// ELIsInHost()
// ============

boolean ELIsInHost(bits(2) el)
    if !HaveVirtHostExt() || ELUsingAArch32(EL2) then
        return FALSE;
    case el of
        when EL3
            return FALSE;
        when EL2
            return EL2Enabled() && HCR_EL2.E2H == '1';
        when EL1
            return FALSE;
        when EL0
            return EL2Enabled() && HCR_EL2.<E2H,TGE> == '11';
        otherwise
            Unreachable();
```

Library pseudocode for shared/functions/system/ELStateUsingAArch32

```
// ELStateUsingAArch32()
// =====================

boolean ELStateUsingAArch32(bits(2) el, boolean secure)
    // See ELStateUsingAArch32K() for description. Must only be called in circumstances where
    // result is valid (typically, that means 'el IN {EL1,EL2,EL3}').
    (known, aarch32) = ELStateUsingAArch32K(el, secure);
    assert known;
    return aarch32;
```

Library pseudocode for shared/functions/system/ELStateUsingAArch32K

```
// ELStateUsingAArch32K()
// ======================

(boolean,boolean) ELStateUsingAArch32K(bits(2) el, boolean secure)
    // Returns (known, aarch32):
    //   'known'   is FALSE for EL0 if the current Exception level is not EL0 and EL1 is
    //             using AArch64, since it cannot determine the state of EL0; TRUE otherwise.
    //   'aarch32' is TRUE if the specified Exception level is using AArch32; FALSE otherwise.
    if !HaveAArch32EL(el) then
        return (TRUE, FALSE);                       // Exception level is using AArch64
    elsif secure && el == EL2 then
        return (TRUE, FALSE);                       // Secure EL2 is using AArch64
    elsif !HaveAArch64() then
        return (TRUE, TRUE);                        // Highest Exception level, and therefore all levels a

    // Remainder of function deals with the interprocessing cases when highest Exception level is using A

    boolean aarch32 = boolean UNKNOWN;
    boolean known = TRUE;

    aarch32_below_el3 = HaveEL(EL3) && SCR_EL3.RW == '0' && (!secure || !HaveSecureEL2Ext() || SCR_EL3.EE
    aarch32_at_el1 = (aarch32_below_el3 || (HaveEL(EL2) &&
                                            ((HaveSecureEL2Ext() && SCR_EL3.EEL2 == '1') || !secure) && H
                                            !(HCR_EL2.E2H == '1' && HCR_EL2.TGE == '1' && HaveVirtHostExt
    if el == EL0 && !aarch32_at_el1 then        // Only know if EL0 using AArch32 from PSTATE
        if PSTATE.EL == EL0 then
            aarch32 = PSTATE.nRW == '1';        // EL0 controlled by PSTATE
        else
            known = FALSE;                      // EL0 state is UNKNOWN
    else
        aarch32 = (aarch32_below_el3 && el != EL3) || (aarch32_at_el1 && el IN {EL1,EL0});

    if !known then aarch32 = boolean UNKNOWN;
    return (known, aarch32);
```

Library pseudocode for shared/functions/system/ELUsingAArch32

```
// ELUsingAArch32()
// ================

boolean ELUsingAArch32(bits(2) el)
    return ELStateUsingAArch32(el, IsSecureBelowEL3());
```

Library pseudocode for shared/functions/system/ELUsingAArch32K

```
// ELUsingAArch32K()
// =================

(boolean,boolean) ELUsingAArch32K(bits(2) el)
    return ELStateUsingAArch32K(el, IsSecureBelowEL3());
```

Library pseudocode for shared/functions/system/EffectiveSCR_EL3_RW

```
// EffectiveSCR_EL3_RW()
// =====================
// Returns effective SCR_EL3.RW value

bit EffectiveSCR_EL3_RW()
    if !HaveAArch64() then
        return '0';
    if !HaveAArch32EL(EL2) && !HaveAArch32EL(EL1) then
        return '1';
    if HaveAArch32EL(EL1) then
        if !HaveAArch32EL(EL2) && SCR_EL3.NS == '1' then
            return '1';
        if HaveSecureEL2Ext() && SCR_EL3.EEL2 == '1' && SCR_EL3.NS == '0' then
            return '1';
    return SCR_EL3.RW;
```

Library pseudocode for shared/functions/system/EffectiveTGE

```
// EffectiveTGE()
// ==============
// Returns effective TGE value

bit EffectiveTGE()
    if EL2Enabled() then
        return if ELUsingAArch32(EL2) then HCR.TGE else HCR_EL2.TGE;
    else
        return '0';        // Effective value of TGE is zero
```

Library pseudocode for shared/functions/system/EndOfInstruction

```
// Terminate processing of the current instruction.
EndOfInstruction();
```

Library pseudocode for shared/functions/system/EnterLowPowerState

```
// PE enters a low-power state.
EnterLowPowerState();
```

Library pseudocode for shared/functions/system/EventRegister

```
bits(1) EventRegister;
```

Library pseudocode for shared/functions/system/ExceptionalOccurrenceTargetState

```
enumeration ExceptionalOccurrenceTargetState {
    AArch32_NonDebugState,
    AArch64_NonDebugState,
    DebugState
};
```

Library pseudocode for shared/functions/system/FIQPending

```
// Returns a tuple indicating if there is any pending physical FIQ
// and if the pending FIQ has superpriority.
(boolean, boolean) FIQPending();
```

Library pseudocode for shared/functions/system/GetAccumulatedFPExceptions

```
// Returns FP exceptions accumulated by the PE.
bits(8) GetAccumulatedFPExceptions();
```

Library pseudocode for shared/functions/system/GetLoadStoreType

```
// Returns the Load/Store Type. Used when a Translation fault,
// Access flag fault, or Permission fault generates a Data Abort.
bits(2) GetLoadStoreType();
```

Library pseudocode for shared/functions/system/GetPSRFromPSTATE

```
// GetPSRFromPSTATE()
// ==================
// Return a PSR value which represents the current PSTATE

bits(N) GetPSRFromPSTATE(ExceptionalOccurrenceTargetState targetELState, integer N)
    if UsingAArch32() && (targetELState IN {AArch32_NonDebugState, DebugState}) then
        assert N == 32;
    else
        assert N == 64;
    bits(N) spsr = Zeros(N);
    spsr<31:28> = PSTATE.<N,Z,C,V>;
    if HavePANExt() then spsr<22> = PSTATE.PAN;
    spsr<20>    = PSTATE.IL;
    if PSTATE.nRW == '1' then                           // AArch32 state
        spsr<27>    = PSTATE.Q;
        spsr<26:25> = PSTATE.IT<1:0>;
        if HaveSSBSExt() then spsr<23> = PSTATE.SSBS;
        if HaveDITExt() then
            if targetELState == AArch32_NonDebugState then
                spsr<21> = PSTATE.DIT;
            else                                        //AArch64_NonDebugState or DebugState
                spsr<24> = PSTATE.DIT;
        if targetELState IN {AArch64_NonDebugState, DebugState} then
            spsr<21> = PSTATE.SS;
        spsr<19:16> = PSTATE.GE;
        spsr<15:10> = PSTATE.IT<7:2>;
        spsr<9>     = PSTATE.E;
        spsr<8:6>   = PSTATE.<A,I,F>;                   // No PSTATE.D in AArch32 state
        spsr<5>     = PSTATE.T;
        assert PSTATE.M<4> == PSTATE.nRW;               // bit [4] is the discriminator
        spsr<4:0>   = PSTATE.M;
    else                                                // AArch64 state
        if HaveMTEExt() then spsr<25> = PSTATE.TCO;
        if HaveDITExt() then spsr<24> = PSTATE.DIT;
        if HaveUAOExt() then spsr<23> = PSTATE.UAO;
        spsr<21>    = PSTATE.SS;
        if HaveFeatNMI() then spsr<13> = PSTATE.ALLINT;
        if HaveSSBSExt() then spsr<12> = PSTATE.SSBS;
        if HaveBTIExt() then spsr<11:10> = PSTATE.BTYPE;
        spsr<9:6>   = PSTATE.<D,A,I,F>;
        spsr<4>     = PSTATE.nRW;
        spsr<3:2>   = PSTATE.EL;
        spsr<0>     = PSTATE.SP;
    return spsr;
```

Library pseudocode for shared/functions/system/HasArchVersion

```
// HasArchVersion()
// ================
// Returns TRUE if the implemented architecture includes the extensions defined in the specified
// architecture version.

boolean HasArchVersion(ArchVersion version)
    return version == ARMv8p0 || boolean IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/functions/system/HaveAArch32

```
// HaveAArch32()
// =============
// Return TRUE if AArch32 state is supported at at least EL0.

boolean HaveAArch32()
    return boolean IMPLEMENTATION_DEFINED "AArch32 state is supported at at least EL0";
```

Library pseudocode for shared/functions/system/HaveAArch32EL

```
// HaveAArch32EL()
// ===============

boolean HaveAArch32EL(bits(2) el)
    // Return TRUE if Exception level 'el' supports AArch32 in this implementation
    if !HaveEL(el) then
        return FALSE;                   // The Exception level is not implemented
    elsif !HaveAArch32() then
        return FALSE;                   // No Exception level can use AArch32
    elsif !HaveAArch64() then
        return TRUE;                    // All Exception levels are using AArch32
    elsif el == HighestEL() then
        return FALSE;                   // The highest Exception level is using AArch64
    elsif el == EL0 then
        return TRUE;                    // EL0 must support using AArch32 if any AArch32
    return boolean IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/functions/system/HaveAArch64

```
// HaveAArch64()
// =============
// Return TRUE if the highest Exception level is using AArch64 state.

boolean HaveAArch64()
    return boolean IMPLEMENTATION_DEFINED "Highest EL using AArch64";
```

Library pseudocode for shared/functions/system/HaveEL

```
// HaveEL()
// ========
// Return TRUE if Exception level 'el' is supported

boolean HaveEL(bits(2) el)
    if el IN {EL1,EL0} then
        return TRUE;                            // EL1 and EL0 must exist
    return boolean IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/functions/system/HaveELUsingSecurityState

```
// HaveELUsingSecurityState()
// ==========================
// Returns TRUE if Exception level 'el' with Security state 'secure' is supported,
// FALSE otherwise.

boolean HaveELUsingSecurityState(bits(2) el, boolean secure)

    case el of
        when EL3
            assert secure;
            return HaveEL(EL3);
        when EL2
            if secure then
                return HaveEL(EL2) && HaveSecureEL2Ext();
            else
                return HaveEL(EL2);
        otherwise
            return (HaveEL(EL3) ||
                    (secure == boolean IMPLEMENTATION_DEFINED "Secure-only implementation"));
```

Library pseudocode for shared/functions/system/HaveFP16Ext

```
// HaveFP16Ext()
// =============
// Return TRUE if FP16 extension is supported

boolean HaveFP16Ext()
    return boolean IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/functions/system/HighestEL

```
// HighestEL()
// ===========
// Returns the highest implemented Exception level.

bits(2) HighestEL()
    if HaveEL(EL3) then
        return EL3;
    elsif HaveEL(EL2) then
        return EL2;
    else
        return EL1;
```

Library pseudocode for shared/functions/system/Hint_DGH

```
// Provides a hint to close any gathering occurring within the micro-architecture.
Hint_DGH();
```

Library pseudocode for shared/functions/system/Hint_WFE

```
// Hint_WFE()
// ==========
// Provides a hint indicating that the PE can enter a low-power state
// and remain there until a wakeup event occurs or, for WFET,  a local
// timeout event is generated when the virtual timer value equals or
// exceeds the supplied threshold value.

Hint_WFE(integer localtimeout, WFxType wfxtype)
    if IsEventRegisterSet() then
        ClearEventRegister();
    elsif HaveFeatWFxT() && LocalTimeoutEvent(localtimeout) then
        // No further operation if the local timeout has expired.
        EndOfInstruction();
    else
        bits(2) target_el;
        trap = FALSE;
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS which may be EL1 or EL2.
            if HaveTWEDExt() then
                sctlr     = SCTLR[];
                trap      = sctlr.nTWE == '0';
                target_el = EL1;
            else
                AArch64.CheckForWFxTrap(EL1, wfxtype);
        if !trap && PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
            // Check for traps described by the Hypervisor.
            if HaveTWEDExt() then
                trap      = HCR_EL2.TWE == '1';
                target_el = EL2;
            else
                AArch64.CheckForWFxTrap(EL2, wfxtype);

        if !trap && HaveEL(EL3) && PSTATE.EL != EL3 then
            // Check for traps described by the Secure Monitor.
            if HaveTWEDExt() then
                trap      = SCR_EL3.TWE == '1';
                target_el = EL3;
            else
                AArch64.CheckForWFxTrap(EL3, wfxtype);

        if trap && PSTATE.EL != EL3 then
            (delay_enabled, delay) = WFETrapDelay(target_el);    // (If trap delay is enabled, Delay amou
            if !WaitForEventUntilDelay(delay_enabled, delay) then
                // Event did not arrive before delay expired
                AArch64.WFxTrap(wfxtype, target_el);             // Trap WFE
        else
            WaitForEvent(localtimeout);
```

Library pseudocode for shared/functions/system/Hint_WFI

```
// Hint_WFI()
// ==========
// Provides a hint indicating that the PE can enter a low-power state and
// remain there until a wakeup event occurs or, for WFIT, a local timeout
// event is generated when the virtual timer value equals or exceeds the
// supplied threshold value.

Hint_WFI(integer localtimeout, WFxType wfxtype)
    if HaveTME() && TSTATE.depth > 0 then
        FailTransaction(TMFailure_ERR, FALSE);

    if InterruptPending() || (HaveFeatWFxT() && LocalTimeoutEvent(localtimeout)) then
        // No further operation if an interrupt is pending or the local timeout has expired.
        EndOfInstruction();
    else
        if PSTATE.EL == EL0 then
            // Check for traps described by the OS.
            AArch64.CheckForWFxTrap(EL1, wfxtype);
        if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
            // Check for traps described by the Hypervisor.
            AArch64.CheckForWFxTrap(EL2, wfxtype);
        if HaveEL(EL3) && PSTATE.EL != EL3 then
            // Check for traps described by the Secure Monitor.
            AArch64.CheckForWFxTrap(EL3, wfxtype);
        WaitForInterrupt(localtimeout);
```

Library pseudocode for shared/functions/system/Hint_Yield

```
// Provides a hint that the task performed by a thread is of low
// importance so that it could yield to improve overall performance.
Hint_Yield();
```

Library pseudocode for shared/functions/system/IRQPending

```
// Returns a tuple indicating if there is any pending physical IRQ
// and if the pending IRQ has superpriority.
(boolean, boolean) IRQPending();
```

Library pseudocode for shared/functions/system/IllegalExceptionReturn

```
// IllegalExceptionReturn()
// ========================

boolean IllegalExceptionReturn(bits(N) spsr)

    // Check for illegal return:
    //   * To an unimplemented Exception level.
    //   * To EL2 in Secure state, when SecureEL2 is not enabled.
    //   * To EL0 using AArch64 state, with SPSR.M[0]==1.
    //   * To AArch64 state with SPSR.M[1]==1.
    //   * To AArch32 state with an illegal value of SPSR.M.
    (valid, target) = ELFromSPSR(spsr);
    if !valid then return TRUE;

    // Check for return to higher Exception level
    if UInt(target) > UInt(PSTATE.EL) then return TRUE;

    spsr_mode_is_aarch32 = (spsr<4> == '1');

    // Check for illegal return:
    //   * To EL1, EL2 or EL3 with register width specified in the SPSR different from the
    //     Execution state used in the Exception level being returned to, as determined by
    //     the SCR_EL3.RW or HCR_EL2.RW bits, or as configured from reset.
    //   * To EL0 using AArch64 state when EL1 is using AArch32 state as determined by the
    //     SCR_EL3.RW or HCR_EL2.RW bits or as configured from reset.
    //   * To AArch64 state from AArch32 state (should be caught by above)
    (known, target_el_is_aarch32) = ELUsingAArch32K(target);
    assert known || (target == EL0 && !ELUsingAArch32(EL1));
    if known && spsr_mode_is_aarch32 != target_el_is_aarch32 then return TRUE;

    // Check for illegal return from AArch32 to AArch64
    if UsingAArch32() && !spsr_mode_is_aarch32 then return TRUE;

    // Check for illegal return to EL1 when HCR.TGE is set and when either of
    // * SecureEL2 is enabled.
    // * SecureEL2 is not enabled and EL1 is in Non-secure state.
    if HaveEL(EL2) && target == EL1 && HCR_EL2.TGE == '1' then
        if (!IsSecureBelowEL3() || IsSecureEL2Enabled()) then return TRUE;
    return FALSE;
```

Library pseudocode for shared/functions/system/InstrSet

```
enumeration InstrSet {InstrSet_A64, InstrSet_A32, InstrSet_T32};
```

Library pseudocode for shared/functions/system/InstructionSynchronizationBarrier

```
InstructionSynchronizationBarrier();
```

Library pseudocode for shared/functions/system/InterruptPending

```
// InterruptPending()
// ==================
// Returns TRUE if there are any pending physical or virtual
// interrupts, and FALSE otherwise.

boolean InterruptPending()
    boolean pending_virtual_interrupt = FALSE;
    (irq_pending, -) = IRQPending();
    (fiq_pending, -) = FIQPending();
    boolean pending_physical_interrupt = (irq_pending || fiq_pending ||
                                          IsPhysicalSErrorPending());

    if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && HCR_EL2.TGE == '0' then
        boolean virq_pending = HCR_EL2.IMO == '1' && (VirtualIRQPending() || HCR_EL2.VI == '1') ;
        boolean vfiq_pending = HCR_EL2.FMO == '1' && (VirtualFIQPending() || HCR_EL2.VF == '1');
        boolean vsei_pending = HCR_EL2.AMO == '1' && (IsVirtualSErrorPending() || HCR_EL2.VSE == '1');
        pending_virtual_interrupt = vsei_pending || virq_pending || vfiq_pending;

    return pending_physical_interrupt || pending_virtual_interrupt;
```

Library pseudocode for shared/functions/system/IsASEInstruction

```
// Returns TRUE if the current instruction is an ASIMD or SVE vector instruction.
boolean IsASEInstruction();
```

Library pseudocode for shared/functions/system/IsCMOWControlledInstruction

```
// When using AArch64, returns TRUE if the current instruction is one of IC IVAU,
// DC CIVAC, DC CIGDVAC, or DC CIGVAC.
// When using AArch32, returns TRUE if the current instruction is ICIMVAU or DCCIMVAC.
boolean IsCMOWControlledInstruction();
```

Library pseudocode for shared/functions/system/IsCurrentSecurityState

```
// IsCurrentSecurityState()
// ========================
// Returns TRUE if the current Security state matches
// the given Security state, and FALSE otherwise.

boolean IsCurrentSecurityState(SecurityState ss)
    return CurrentSecurityState() == ss;
```

Library pseudocode for shared/functions/system/IsEventRegisterSet

```
// IsEventRegisterSet()
// ====================
// Return TRUE if the Event Register of this PE is set, and FALSE if it is clear.

boolean IsEventRegisterSet()
    return EventRegister == '1';
```

Library pseudocode for shared/functions/system/IsHighestEL

```
// IsHighestEL()
// =============
// Returns TRUE if given exception level is the highest exception level implemented

boolean IsHighestEL(bits(2) el)
    return HighestEL() == el;
```

Library pseudocode for shared/functions/system/IsInHost

```
// IsInHost()
// ==========

boolean IsInHost()
    return ELIsInHost(PSTATE.EL);
```

Library pseudocode for shared/functions/system/IsPhysicalSErrorPending

```
// Returns TRUE if a physical SError interrupt is pending.
boolean IsPhysicalSErrorPending();
```

Library pseudocode for shared/functions/system/IsSErrorEdgeTriggered

```
// IsSErrorEdgeTriggered()
// =======================
// Returns TRUE if the physical SError interrupt is edge-triggered
// and FALSE otherwise.

boolean IsSErrorEdgeTriggered(bits(2) target_el, bits(25) syndrome)
    if HaveRASExt() then
        if HaveDoubleFaultExt() then
            return TRUE;

        if ELUsingAArch32(target_el) then
            if syndrome<11:10> != '00' then
                // AArch32 and not Uncontainable.
                return TRUE;
        else
            if syndrome<24> == '0' && syndrome<5:0> != '000000' then
                // AArch64 and neither IMPLEMENTATION DEFINED syndrome nor Uncategorized.
                return TRUE;
    return boolean IMPLEMENTATION_DEFINED "Edge-triggered SError";
```

Library pseudocode for shared/functions/system/IsSecure

```
// IsSecure()
// ==========
// Returns TRUE if current Exception level is in Secure state.

boolean IsSecure()
    if HaveEL(EL3) && !UsingAArch32() && PSTATE.EL == EL3 then
        return TRUE;
    elsif HaveEL(EL3) && UsingAArch32() && PSTATE.M == M32_Monitor then
        return TRUE;
    return IsSecureBelowEL3();
```

Library pseudocode for shared/functions/system/IsSecureBelowEL3

```
// IsSecureBelowEL3()
// ==================
// Return TRUE if an Exception level below EL3 is in Secure state
// or would be following an exception return to that level.
//
// Differs from IsSecure in that it ignores the current EL or Mode
// in considering security state.
// That is, if at AArch64 EL3 or in AArch32 Monitor mode, whether an
// exception return would pass to Secure or Non-secure state.

boolean IsSecureBelowEL3()
    if HaveEL(EL3) then
        return SCR_GEN[].NS == '0';
    elsif HaveEL(EL2) && (!HaveSecureEL2Ext() || !HaveAArch64()) then
        // If Secure EL2 is not an architecture option then we must be Non-secure.
        return FALSE;
    else
        // TRUE if processor is Secure or FALSE if Non-secure.
        return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";
```

Library pseudocode for shared/functions/system/IsSecureEL2Enabled

```
// IsSecureEL2Enabled()
// ====================
// Returns TRUE if Secure EL2 is enabled, FALSE otherwise.

boolean IsSecureEL2Enabled()
    if HaveEL(EL2) && HaveSecureEL2Ext() then
        if HaveEL(EL3) then
            if !ELUsingAArch32(EL3) && SCR_EL3.EEL2 == '1' then
                return TRUE;
            else
                return FALSE;
        else
            return SecureOnlyImplementation();
    else
        return FALSE;
```

Library pseudocode for shared/functions/system/IsSynchronizablePhysicalSErrorPending

```
// Returns TRUE if a synchronizable physical SError interrupt is pending.
boolean IsSynchronizablePhysicalSErrorPending();
```

Library pseudocode for shared/functions/system/IsVirtualSErrorPending

```
// Returns TRUE if a virtual SError interrupt is pending.
boolean IsVirtualSErrorPending();
```

Library pseudocode for shared/functions/system/LocalTimeoutEvent

```
// Returns TRUE if CNTVCT_EL0 equals or exceeds the localtimeout value.
boolean LocalTimeoutEvent(integer localtimeout);
```

Library pseudocode for shared/functions/system/Mode_Bits

```
constant bits(5) M32_User    = '10000';
constant bits(5) M32_FIQ     = '10001';
constant bits(5) M32_IRQ     = '10010';
constant bits(5) M32_Svc     = '10011';
constant bits(5) M32_Monitor = '10110';
constant bits(5) M32_Abort   = '10111';
constant bits(5) M32_Hyp     = '11010';
constant bits(5) M32_Undef   = '11011';
constant bits(5) M32_System  = '11111';
```

Library pseudocode for shared/functions/system/NonSecureOnlyImplementation

```
// NonSecureOnlyImplementation()
// =============================
// Returns TRUE if the security state is always Non-secure for this implementation.

boolean NonSecureOnlyImplementation()
    return boolean IMPLEMENTATION_DEFINED "Non-secure only implementation";
```

Library pseudocode for shared/functions/system/PLOfEL

```
// PLOfEL()
// ========

PrivilegeLevel PLOfEL(bits(2) el)
    case el of
        when EL3  return if !HaveAArch64() then PL1 else PL3;
        when EL2  return PL2;
        when EL1  return PL1;
        when EL0  return PL0;
```

Library pseudocode for shared/functions/system/PSTATE

```
ProcState PSTATE;
```

Library pseudocode for shared/functions/system/PhysicalCountInt

```
// PhysicalCountInt()
// ==================
// Returns the integral part of physical count value of the System counter.

bits(64) PhysicalCountInt()
    return PhysicalCount<87:24>;
```

Library pseudocode for shared/functions/system/PrivilegeLevel

```
enumeration PrivilegeLevel {PL3, PL2, PL1, PL0};
```

Library pseudocode for shared/functions/system/ProcState

```
type ProcState is (
    bits (1) N,        // Negative condition flag
    bits (1) Z,        // Zero condition flag
    bits (1) C,        // Carry condition flag
    bits (1) V,        // Overflow condition flag
    bits (1) D,        // Debug mask bit                      [AArch64 only]
    bits (1) A,        // SError interrupt mask bit
    bits (1) I,        // IRQ mask bit
    bits (1) F,        // FIQ mask bit
    bits (1) PAN,      // Privileged Access Never Bit         [v8.1]
    bits (1) UAO,      // User Access Override                 [v8.2]
    bits (1) DIT,      // Data Independent Timing              [v8.4]
    bits (1) TCO,      // Tag Check Override                  [v8.5, AArch64 only]
    bits (2) BTYPE,    // Branch Type                         [v8.5]
    bits (1) ZA,       // Accumulation array enabled          [SME]
    bits (1) SM,       // Streaming SVE mode enabled          [SME]
    bits (1) ALLINT,   // Interrupt mask bit
    bits (1) SS,       // Software step bit
    bits (1) IL,       // Illegal Execution state bit
    bits (2) EL,       // Exception level
    bits (1) nRW,      // not Register Width: 0=64, 1=32
    bits (1) SP,       // Stack pointer select: 0=SP0, 1=SPx [AArch64 only]
    bits (1) Q,        // Cumulative saturation flag          [AArch32 only]
    bits (4) GE,       // Greater than or Equal flags         [AArch32 only]
    bits (1) SSBS,     // Speculative Store Bypass Safe
    bits (8) IT,       // If-then bits, RES0 in CPSR          [AArch32 only]
    bits (1) J,        // J bit, RES0                         [AArch32 only, RES0 in SPSR and CPSR]
    bits (1) T,        // T32 bit, RES0 in CPSR               [AArch32 only]
    bits (1) E,        // Endianness bit                      [AArch32 only]
    bits (5) M         // Mode field                          [AArch32 only]
)
```

Library pseudocode for shared/functions/system/RestoredITBits

```
// RestoredITBits()
// ================
// Get the value of PSTATE.IT to be restored on this exception return.

bits(8) RestoredITBits(bits(N) spsr)
    it = spsr<15:10,26:25>;

    // When PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the IT bits are each set
    // to zero or copied from the SPSR.
    if PSTATE.IL == '1' then
        if ConstrainUnpredictableBool(Unpredictable_ILZEROIT) then return '00000000';
        else return it;

    // The IT bits are forced to zero when they are set to a reserved value.
    if !IsZero(it<7:4>) && IsZero(it<3:0>) then
        return '00000000';

    // The IT bits are forced to zero when returning to A32 state, or when returning to an EL
    // with the ITD bit set to 1, and the IT bits are describing a multi-instruction block.
    itd = if PSTATE.EL == EL2 then HSCTLR.ITD else SCTLR.ITD;
    if (spsr<5> == '0' && !IsZero(it)) || (itd == '1' && !IsZero(it<2:0>)) then
        return '00000000';
    else
        return it;
```

Library pseudocode for shared/functions/system/SCRType

```
type SCRType;
```

Library pseudocode for shared/functions/system/SCR_GEN

```
// SCR_GEN[]
// =========

SCRType SCR_GEN[]
    // AArch32 secure & AArch64 EL3 registers are not architecturally mapped
    assert HaveEL(EL3);
    bits(64) r;
    if !HaveAArch64() then
        r = ZeroExtend(SCR, 64);
    else
        r = SCR_EL3;
    return r;
```

Library pseudocode for shared/functions/system/SecureOnlyImplementation

```
// SecureOnlyImplementation()
// ==========================
// Returns TRUE if the security state is always Secure for this implementation.

boolean SecureOnlyImplementation()
    return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";
```

Library pseudocode for shared/functions/system/SecurityState

```
enumeration SecurityState {
    SS_NonSecure,
    SS_Root,
    SS_Realm,
    SS_Secure
};
```

Library pseudocode for shared/functions/system/SecurityStateAtEL

```
// SecurityStateAtEL()
// ===================
// Returns the effective security state at the exception level based off current settings.

SecurityState SecurityStateAtEL(bits(2) EL)
    if HaveRME() then
        if EL == EL3 then return SS_Root;
        case SCR_EL3.<NSE, NS> of
            when '00' return SS_Secure;
            when '01' return SS_NonSecure;
            when '11' return SS_Realm;
    if !HaveEL(EL3) then
        if SecureOnlyImplementation() then
            return SS_Secure;
        else
            return SS_NonSecure;
    elsif EL == EL3 then
        return SS_Secure;
    else
        // For EL2 call only when EL2 is enabled in current security state
        assert(EL != EL2 || EL2Enabled());
        if !ELUsingAArch32(EL3) then
            return if SCR_EL3.NS == '1' then SS_NonSecure else SS_Secure;
        else
            return if SCR.NS == '1' then SS_NonSecure else SS_Secure;
```

Library pseudocode for shared/functions/system/SendEvent

```
// Signal an event to all PEs in a multiprocessor system to set their Event Registers.
// When a PE executes the SEV instruction, it causes this function to be executed.
SendEvent();
```

Library pseudocode for shared/functions/system/SendEventLocal

```
// SendEventLocal()
// ================
// Set the local Event Register of this PE.
// When a PE executes the SEVL instruction, it causes this function to be executed.

SendEventLocal()
    EventRegister = '1';
    return;
```

Library pseudocode for shared/functions/system/SetAccumulatedFPExceptions

```
// Stores FP Exceptions accumulated by the PE.
SetAccumulatedFPExceptions(bits(8) accumulated_exceptions);
```

Library pseudocode for shared/functions/system/SetPSTATEFromPSR

```
// SetPSTATEFromPSR()
// ==================

SetPSTATEFromPSR(bits(N) spsr)
    boolean illegal_psr_state = IllegalExceptionReturn(spsr);
    SetPSTATEFromPSR(spsr, illegal_psr_state);

// SetPSTATEFromPSR()
// ==================
// Set PSTATE based on a PSR value

SetPSTATEFromPSR(bits(N) spsr_in, boolean illegal_psr_state)
    bits(N) spsr = spsr_in;
    boolean from_aarch64 = !UsingAArch32();
    assert N == (if from_aarch64 then 64 else 32);
    PSTATE.SS = DebugExceptionReturnSS(spsr);
    ShouldAdvanceSS = FALSE;
    if illegal_psr_state then
        PSTATE.IL = '1';
        if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
        if HaveBTIExt() then PSTATE.BTYPE = bits(2) UNKNOWN;
        if HaveUAOExt() then PSTATE.UAO = bit UNKNOWN;
        if HaveDITExt() then PSTATE.DIT = bit UNKNOWN;
        if HaveMTEExt() then PSTATE.TCO = bit UNKNOWN;
    else
        // State that is reinstated only on a legal exception return
        PSTATE.IL = spsr<20>;
        if spsr<4> == '1' then                    // AArch32 state
            AArch32.WriteMode(spsr<4:0>);         // Sets PSTATE.EL correctly
            if HaveSSBSExt() then PSTATE.SSBS = spsr<23>;
        else                                      // AArch64 state
            PSTATE.nRW = '0';
            PSTATE.EL  = spsr<3:2>;
            PSTATE.SP  = spsr<0>;
            if HaveBTIExt() then PSTATE.BTYPE = spsr<11:10>;
            if HaveSSBSExt() then PSTATE.SSBS = spsr<12>;
            if HaveUAOExt() then PSTATE.UAO = spsr<23>;
            if HaveDITExt() then PSTATE.DIT = spsr<24>;
            if HaveMTEExt() then PSTATE.TCO = spsr<25>;

    // If PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the T bit is set to zero or
    // copied from SPSR.
    if PSTATE.IL == '1' && PSTATE.nRW == '1' then
        if ConstrainUnpredictableBool(Unpredictable_ILZEROT) then spsr<5> = '0';

    // State that is reinstated regardless of illegal exception return
    PSTATE.<N,Z,C,V> = spsr<31:28>;
    if HavePANExt() then PSTATE.PAN = spsr<22>;
    if PSTATE.nRW == '1' then                     // AArch32 state
        PSTATE.Q         = spsr<27>;
        PSTATE.IT        = RestoredITBits(spsr);
        ShouldAdvanceIT  = FALSE;
        if HaveDITExt() then PSTATE.DIT = (if (Restarting() || from_aarch64) then spsr<24> else spsr<21>)
        PSTATE.GE        = spsr<19:16>;
        PSTATE.E         = spsr<9>;
        PSTATE.<A,I,F>   = spsr<8:6>;             // No PSTATE.D in AArch32 state
        PSTATE.T         = spsr<5>;               // PSTATE.J is RES0
    else                                          // AArch64 state
        if HaveFeatNMI() then PSTATE.ALLINT = spsr<13>;
        PSTATE.<D,A,I,F> = spsr<9:6>;             // No PSTATE.<Q,IT,GE,E,T> in AArch64 state
    return;
```

Library pseudocode for shared/functions/system/ShouldAdvanceIT

```
boolean ShouldAdvanceIT;
```

Library pseudocode for shared/functions/system/ShouldAdvanceSS

```
boolean ShouldAdvanceSS;
```

Library pseudocode for shared/functions/system/SpeculationBarrier

```
SpeculationBarrier();
```

Library pseudocode for shared/functions/system/SynchronizeContext

```
SynchronizeContext();
```

Library pseudocode for shared/functions/system/SynchronizeErrors

```
// Implements the error synchronization event.
SynchronizeErrors();
```

Library pseudocode for shared/functions/system/TakeUnmaskedPhysicalSErrorInterrupts

```
// Take any pending unmasked physical SError interrupt.
TakeUnmaskedPhysicalSErrorInterrupts(boolean iesb_req);
```

Library pseudocode for shared/functions/system/TakeUnmaskedSErrorInterrupts

```
// Take any pending unmasked physical SError interrupt or unmasked virtual SError
// interrupt.
TakeUnmaskedSErrorInterrupts();
```

Library pseudocode for shared/functions/system/ThisInstr

```
bits(32) ThisInstr();
```

Library pseudocode for shared/functions/system/ThisInstrLength

```
integer ThisInstrLength();
```

Library pseudocode for shared/functions/system/Unreachable

```
Unreachable()
    assert FALSE;
```

Library pseudocode for shared/functions/system/UsingAArch32

```
// UsingAArch32()
// ==============
// Return TRUE if the current Exception level is using AArch32, FALSE if using AArch64.

boolean UsingAArch32()
    boolean aarch32 = (PSTATE.nRW == '1');
    if !HaveAArch32() then assert !aarch32;
    if !HaveAArch64() then assert aarch32;
    return aarch32;
```

Library pseudocode for shared/functions/system/VirtualFIQPending

```
// Returns TRUE if there is any pending virtual FIQ.
boolean VirtualFIQPending();
```

Library pseudocode for shared/functions/system/VirtualIRQPending

```
// Returns TRUE if there is any pending virtual IRQ.
boolean VirtualIRQPending();
```

Library pseudocode for shared/functions/system/WFxType

```
enumeration WFxType {WFxType_WFE, WFxType_WFI, WFxType_WFET, WFxType_WFIT};
```

Library pseudocode for shared/functions/system/WaitForEvent

```
// WaitForEvent()
// ==============
// PE optionally suspends execution until one of the following occurs:
// - A WFE wakeup event.
// - A reset.
// - The implementation chooses to resume execution.
// - A Wait for Event with Timeout (WFET) is executing, and a local timeout event occurs
// It is IMPLEMENTATION DEFINED whether restarting execution after the period of
// suspension causes the Event Register to be cleared.

WaitForEvent(integer localtimeout)
    if !(IsEventRegisterSet() || (HaveFeatWFxT() && LocalTimeoutEvent(localtimeout))) then
        EnterLowPowerState();
    return;
```

Library pseudocode for shared/functions/system/WaitForInterrupt

```
// WaitForInterrupt()
// ==================
// PE optionally suspends execution until one of the following occurs:
// - A WFI wakeup event.
// - A reset.
// - The implementation chooses to resume execution.
// - A Wait for Interrupt with Timeout (WFIT) is executing, and a local timeout event occurs.

WaitForInterrupt(integer localtimeout)
    if !(HaveFeatWFxT() && LocalTimeoutEvent(localtimeout)) then
        EnterLowPowerState();
    return;
```

Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictable

```
// Return the appropriate Constraint result to control the caller's behavior.
// The return value is IMPLEMENTATION DEFINED within a permitted list for each
// UNPREDICTABLE case.
// (The permitted list is determined by an assert or case statement at the call site.)
Constraint ConstrainUnpredictable(Unpredictable which);
```

Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableBits

```
// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN.
// If the result is Constraint_UNKNOWN then the function also returns UNKNOWN value, but that
// value is always an allocated value; that is, one for which the behavior is not itself
// CONSTRAINED.
(Constraint,bits(width)) ConstrainUnpredictableBits(Unpredictable which, integer width);
```

Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableBool

```
// This is a variant of the ConstrainUnpredictable function where the result is either
// Constraint_TRUE or Constraint_FALSE.
boolean ConstrainUnpredictableBool(Unpredictable which);
```

Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableInteger

```
// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN.
// If the result is Constraint_UNKNOWN then the function also returns an UNKNOWN
// value in the range low to high, inclusive.
(Constraint,integer) ConstrainUnpredictableInteger(integer low, integer high,
                                                   Unpredictable which);
```

Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableProcedure

```
// This is a variant of ConstrainUnpredictable that implements a Constrained
// Unpredictable behavior for a given Unpredictable situation.
// The behavior is within permitted behaviors for a given Unpredictable situation,
// these are documented in the textual part of the architecture specification.
//
// This function is expected to be refined in an IMPLEMENTATION DEFINED manner.
// The details of possible outcomes may not be present in the code and must be interpreted
// for each use with respect to the CONSTRAINED UNPREDICTABLE specifications
// for the specific area.
ConstrainUnpredictableProcedure(Unpredictable which);
```

Library pseudocode for shared/functions/unpredictable/Constraint

```
enumeration Constraint    {// General
                           Constraint_NONE,              // Instruction executes with
                                                         // no change or side-effect
                                                         // to its described behavior
                           Constraint_UNKNOWN,           // Destination register
                                                         // has UNKNOWN value
                           Constraint_UNDEF,             // Instruction is UNDEFINED
                           Constraint_UNDEFEL0,          // Instruction is UNDEFINED at EL0 only
                           Constraint_NOP,               // Instruction executes as NOP
                           Constraint_TRUE,
                           Constraint_FALSE,
                           Constraint_DISABLED,
                           Constraint_UNCOND,            // Instruction executes unconditionally
                           Constraint_COND,              // Instruction executes conditionally
                           Constraint_ADDITIONAL_DECODE, // Instruction executes
                                                         // with additional decode
                           // Load-store
                           Constraint_WBSUPPRESS,
                           Constraint_FAULT,
                           Constraint_LIMITED_ATOMICITY, // Accesses are not
                                                         // single-copy atomic
                                                         // above the byte level
                           Constraint_NVNV1_00,
                           Constraint_NVNV1_01,
                           Constraint_NVNV1_11,
                           Constraint_EL1TIMESTAMP,      // Constrain to Virtual Timestamp
                           Constraint_EL2TIMESTAMP,      // Constrain to Virtual Timestamp
                           Constraint_OSH,               // Constrain to Outer Shareable
                           Constraint_ISH,               // Constrain to Inner Shareable
                           Constraint_NSH,               // Constrain to Nonshareable

                           Constraint_NC,                // Constrain to Noncacheable
                           Constraint_WT,                // Constrain to Writethrough
                           Constraint_WB,                // Constrain to Writeback

                           // IPA too large
                           Constraint_FORCE, Constraint_FORCENOSLCHECK,
                           // PMSCR_PCT reserved values select Virtual timestamp
                           Constraint_PMSCR_PCT_VIRT};
```

Library pseudocode for shared/functions/unpredictable/Unpredictable

```
enumeration Unpredictable {
                            // VMSR on MVFR
                            Unpredictable_VMSR,
                            // Writeback/transfer register overlap (load)
                            Unpredictable_WBOVERLAPLD,
                            // Writeback/transfer register overlap (store)
                            Unpredictable_WBOVERLAPST,
                            // Load Pair transfer register overlap
                            Unpredictable_LDPOVERLAP,
                            // Store-exclusive base/status register overlap
                            Unpredictable_BASEOVERLAP,
                            // Store-exclusive data/status register overlap
                            Unpredictable_DATAOVERLAP,
                            // Load-store alignment checks
                            Unpredictable_DEVPAGE2,
                            // Instruction fetch from Device memory
                            Unpredictable_INSTRDEVICE,
                            // Reserved CPACR value
                            Unpredictable_RESCPACR,
                            // Reserved MAIR value
                            Unpredictable_RESMAIR,
                            // Effect of SCTLR_ELx.C on Tagged attribute
                            Unpredictable_S1CTAGGED,
                            // Reserved Stage 2 MemAttr value
                            Unpredictable_S2RESMEMATTR,
                            // Reserved TEX:C:B value
                            Unpredictable_RESTEXCB,
                            // Reserved PRRR value
                            Unpredictable_RESPRRR,
                            // Reserved DACR field
                            Unpredictable_RESDACR,
                            // Reserved VTCR.S value
                            Unpredictable_RESVTCRS,
                            // Reserved TCR.TnSZ value
                            Unpredictable_RESTnSZ,
                            // Reserved SCTLR_ELx.TCF value
                            Unpredictable_RESTCF,
                            // Tag stored to Device memory
                            Unpredictable_DEVICETAGSTORE,
                            // Out-of-range TCR.TnSZ value
                            Unpredictable_OORTnSZ,
                            // IPA size exceeds PA size
                            Unpredictable_LARGEIPA,
                            // Syndrome for a known-passing conditional A32 instruction
                            Unpredictable_ESRCONDPASS,
                            // Illegal State exception: zero PSTATE.IT
                            Unpredictable_ILZEROIT,
                            // Illegal State exception: zero PSTATE.T
                            Unpredictable_ILZEROT,
                            // Debug: prioritization of Vector Catch
                            Unpredictable_BPVECTORCATCHPRI,
                            // Debug Vector Catch: match on 2nd halfword
                            Unpredictable_VCMATCHHALF,
                            // Debug Vector Catch: match on Data Abort
                            // or Prefetch abort
                            Unpredictable_VCMATCHDAPA,
                            // Debug watchpoints: non-zero MASK and non-ones BAS
                            Unpredictable_WPMASKANDBAS,
                            // Debug watchpoints: non-contiguous BAS
                            Unpredictable_WPBASCONTIGUOUS,
                            // Debug watchpoints: reserved MASK
                            Unpredictable_RESWPMASK,
                            // Debug watchpoints: non-zero MASKed bits of address
                            Unpredictable_WPMASKEDBITS,
                            // Debug breakpoints and watchpoints: reserved control bits
                            Unpredictable_RESBPWPCTRL,
                            // Debug breakpoints: not implemented
                            Unpredictable_BPNOTIMPL,
                            // Debug breakpoints: reserved type
                            Unpredictable_RESBPTYPE,
```

```
                        // Debug breakpoints: not-context-aware breakpoint
                        Unpredictable_BPNOTCTXCMP,
                        // Debug breakpoints: match on 2nd halfword of instruction
                        Unpredictable_BPMATCHHALF,
                        // Debug breakpoints: mismatch on 2nd halfword of instruction
                        Unpredictable_BPMISMATCHHALF,
                        // Debug: restart to a misaligned AArch32 PC value
                        Unpredictable_RESTARTALIGNPC,
                        // Debug: restart to a not-zero-extended AArch32 PC value
                        Unpredictable_RESTARTZEROUPPERPC,
                        // Zero top 32 bits of X registers in AArch32 state
                        Unpredictable_ZEROUPPER,
                        // Zero top 32 bits of PC on illegal return to
                        // AArch32 state
                        Unpredictable_ERETZEROUPPERPC,
                        // Force address to be aligned when interworking
                        // branch to A32 state
                        Unpredictable_A32FORCEALIGNPC,
                        // SMC disabled
                        Unpredictable_SMD,
                        // FF speculation
                        Unpredictable_NONFAULT,
                        // Zero top bits of Z registers in EL change
                        Unpredictable_SVEZEROUPPER,
                        // Load mem data in NF loads
                        Unpredictable_SVELDNFDATA,
                        // Write zeros in NF loads
                        Unpredictable_SVELDNFZERO,
                        // SP alignment fault when predicate is all zero
                        Unpredictable_CHECKSPNONEACTIVE,
                        // Zero top bits of ZA registers in EL change
                        Unpredictable_SMEZEROUPPER,
                        // HCR_EL2.<NV,NV1> == '01'
                        Unpredictable_NVNV1,
                        // Reserved shareability encoding
                        Unpredictable_Shareability,
                        // Access Flag Update by HW
                        Unpredictable_AFUPDATE,
                        // Dirty Bit State Update by HW
                        Unpredictable_DBUPDATE,
                        // Consider SCTLR[].IESB in Debug state
                        Unpredictable_IESBinDebug,
                        // Bad settings for PMSFCR_EL1/PMSEVFR_EL1/PMSLATFR_EL1
                        Unpredictable_BADPMSFCR,
                        // Zero saved BType value in SPSR_ELx/DPSR_EL0
                        Unpredictable_ZEROBTYPE,
                        // Timestamp constrained to virtual or physical
                        Unpredictable_EL2TIMESTAMP,
                        Unpredictable_EL1TIMESTAMP,
                         // Reserved MDCR_EL3.<NSTBE,NSTB> or MDCR_EL3.<NSPBE,NSPB> value
                         Unpredictable_RESERVEDNSxB,
                        // WFET or WFIT instruction in Debug state
                        Unpredictable_WFxTDEBUG,
                        // Address does not support LS64 instructions
                        Unpredictable_LS64UNSUPPORTED,
                        // Misaligned exclusives, atomics, acquire/release
                        // to region that is not Normal Cacheable WB
                        Unpredictable_MISALIGNEDATOMIC,
                        // Clearing DCC/ITR sticky flags when instruction is in flight
                        Unpredictable_CLEARERRITEZERO,
                        // ALUEXCEPTIONRETURN when in user/system mode in
                        // A32 instructions
                        Unpredictable_ALUEXCEPTIONRETURN,
                        // Trap to register in debug state are ignored
                        Unpredictable_IGNORETRAPINDEBUG,
                        // Compare DBGBVR.RESS for BP/WP
                        Unpredictable_DBGxVR_RESS,
                        // Inaccessible event counter
                        Unpredictable_PMUEVENTCOUNTER,
                        // Reserved PMSCR.PCT behaviour.
```

```
                          Unpredictable_PMSCR_PCT,
                          // Generate BRB_FILTRATE event on BRB injection
                          Unpredictable_BRBFILTRATE,
                         // Operands for CPY*/SET* instructions overlap or
                         // use 0b11111 as a register specifier
                         Unpredictable_MOPSOVERLAP31
};
```

Library pseudocode for shared/functions/vector/AdvSIMDExpandImm

```
// AdvSIMDExpandImm()
// ==================

bits(64) AdvSIMDExpandImm(bit op, bits(4) cmode, bits(8) imm8)
    bits(64) imm64;
    case cmode<3:1> of
        when '000'
            imm64 = Replicate(Zeros(24):imm8, 2);
        when '001'
            imm64 = Replicate(Zeros(16):imm8:Zeros(8), 2);
        when '010'
            imm64 = Replicate(Zeros(8):imm8:Zeros(16), 2);
        when '011'
            imm64 = Replicate(imm8:Zeros(24), 2);
        when '100'
            imm64 = Replicate(Zeros(8):imm8, 4);
        when '101'
            imm64 = Replicate(imm8:Zeros(8), 4);
        when '110'
            if cmode<0> == '0' then
                imm64 = Replicate(Zeros(16):imm8:Ones(8), 2);
            else
                imm64 = Replicate(Zeros(8):imm8:Ones(16), 2);
        when '111'
            if cmode<0> == '0' && op == '0' then
                imm64 = Replicate(imm8, 8);
            if cmode<0> == '0' && op == '1' then
                imm8a = Replicate(imm8<7>, 8); imm8b = Replicate(imm8<6>, 8);
                imm8c = Replicate(imm8<5>, 8); imm8d = Replicate(imm8<4>, 8);
                imm8e = Replicate(imm8<3>, 8); imm8f = Replicate(imm8<2>, 8);
                imm8g = Replicate(imm8<1>, 8); imm8h = Replicate(imm8<0>, 8);
                imm64 = imm8a:imm8b:imm8c:imm8d:imm8e:imm8f:imm8g:imm8h;
            if cmode<0> == '1' && op == '0' then
                imm32 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,5):imm8<5:0>:Zeros(19);
                imm64 = Replicate(imm32, 2);
            if cmode<0> == '1' && op == '1' then
                if UsingAArch32() then ReservedEncoding();
                imm64 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,8):imm8<5:0>:Zeros(48);

    return imm64;
```

Library pseudocode for shared/functions/vector/MatMulAdd

```
// MatMulAdd()
// ===========
//
// Signed or unsigned 8-bit integer matrix multiply and add to 32-bit integer matrix
// result[2, 2] = addend[2, 2] + (op1[2, 8] * op2[8, 2])

bits(N) MatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, boolean op1_unsigned, boolean op2_unsigned)
    assert N == 128;

    bits(N)  result;
    bits(32) sum;
    integer  prod;

    for i = 0 to 1
        for j = 0 to 1
            sum = Elem[addend, 2*i + j, 32];
            for k = 0 to 7
                prod = Int(Elem[op1, 8*i + k, 8], op1_unsigned) * Int(Elem[op2, 8*j + k, 8], op2_unsigned
                sum  = sum + prod;
            Elem[result, 2*i + j, 32] = sum;

    return result;
```

Library pseudocode for shared/functions/vector/PolynomialMult

```
// PolynomialMult()
// ================

bits(M+N) PolynomialMult(bits(M) op1, bits(N) op2)
    result = Zeros(M+N);
    extended_op2 = ZeroExtend(op2, M+N);
    for i=0 to M-1
        if op1<i> == '1' then
            result = result EOR LSL(extended_op2, i);
    return result;
```

Library pseudocode for shared/functions/vector/SatQ

```
// SatQ()
// ======

(bits(N), boolean) SatQ(integer i, integer N, boolean unsigned)
    (result, sat) = if unsigned then UnsignedSatQ(i, N) else SignedSatQ(i, N);
    return (result, sat);
```

Library pseudocode for shared/functions/vector/SignedSatQ

```
// SignedSatQ()
// ============

(bits(N), boolean) SignedSatQ(integer i, integer N)
    integer result;
    boolean saturated;
    if i > 2^(N-1) - 1 then
        result = 2^(N-1) - 1;  saturated = TRUE;
    elsif i < -(2^(N-1)) then
        result = -(2^(N-1));  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<N-1:0>, saturated);
```

Library pseudocode for shared/functions/vector/UnsignedRSqrtEstimate

```
// UnsignedRSqrtEstimate()
// =======================

bits(N) UnsignedRSqrtEstimate(bits(N) operand)
    assert N == 32;
    bits(N) result;
    if operand<N-1:N-2> == '00' then  // Operands <= 0x3FFFFFFF produce 0xFFFFFFFF
        result = Ones(N);
    else
        // input is in the range 0x40000000 .. 0xffffffff representing [0.25 .. 1.0)
        // estimate is in the range 256 .. 511 representing [1.0 .. 2.0)
        increasedprecision = FALSE;
        estimate = RecipSqrtEstimate(UInt(operand<31:23>), increasedprecision);
        // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
        result = estimate<8:0> : Zeros(N-9);

    return result;
```

Library pseudocode for shared/functions/vector/UnsignedRecipEstimate

```
// UnsignedRecipEstimate()
// =======================

bits(N) UnsignedRecipEstimate(bits(N) operand)
    assert N == 32;
    bits(N) result;
    if operand<N-1> == '0' then  // Operands <= 0x7FFFFFFF produce 0xFFFFFFFF
        result = Ones(N);
    else
        // input is in the range 0x80000000 .. 0xffffffff representing [0.5 .. 1.0)

        // estimate is in the range 256 to 511 representing [1.0 .. 2.0)
        increasedprecision = FALSE;
        estimate = RecipEstimate(UInt(operand<31:23>), increasedprecision);

        // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
        result = estimate<8:0> : Zeros(N-9);

    return result;
```

Library pseudocode for shared/functions/vector/UnsignedSatQ

```
// UnsignedSatQ()
// ==============

(bits(N), boolean) UnsignedSatQ(integer i, integer N)
    integer result;
    boolean saturated;
    if i > 2^N - 1 then
        result = 2^N - 1;  saturated = TRUE;
    elsif i < 0 then
        result = 0;  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<N-1:0>, saturated);
```

Library pseudocode for shared/trace/Common/GetTimestamp

```
// GetTimestamp()
// ==============
// Returns the Timestamp depending on the type

bits(64) GetTimestamp(TimeStamp timeStampType)
    case timeStampType of
        when TimeStamp_Physical
            return PhysicalCountInt();
        when TimeStamp_Virtual
            return PhysicalCountInt() - CNTVOFF_EL2;
        when TimeStamp_OffsetPhysical
            bits(64) physoff = if PhysicalOffsetIsValid() then CNTPOFF_EL2 else Zeros();
            return PhysicalCountInt() - physoff;
        when TimeStamp_None
            return Zeros(64);
        when TimeStamp_CoreSight
            return bits(64) IMPLEMENTATION_DEFINED "CoreSight timestamp";
        otherwise
            Unreachable();
```

Library pseudocode for shared/trace/Common/PhysicalOffsetIsValid

```
// PhysicalOffsetIsValid()
// =======================
// Returns whether the Physical offset for the timestamp is valid

boolean PhysicalOffsetIsValid()
    if !HaveAArch64() then
        return FALSE;
    elsif !HaveEL(EL2) || !HaveECVExt() then
        return FALSE;
    elsif HaveEL(EL3) && SCR_EL3.NS == '1' && EffectiveSCR_EL3_RW() == '0' then
        return FALSE;
    elsif HaveEL(EL3) && SCR_EL3.ECVEn == '0' then
        return FALSE;
    elsif CNTHCTL_EL2.ECV == '0' then
        return FALSE;
    else
        return TRUE;
```

Library pseudocode for shared/trace/TraceBuffer/TraceBufferEnabled

```
// TraceBufferEnabled()
// ====================

boolean TraceBufferEnabled()
    if !HaveTraceBufferExtension() || TRBLIMITR_EL1.E == '0' then
        return FALSE;
    if !SelfHostedTraceEnabled() then
        return FALSE;
    (-, el) = TraceBufferOwner();
    return !ELUsingAArch32(el);
```

Library pseudocode for shared/trace/TraceBuffer/TraceBufferOwner

```
// TraceBufferOwner()
// ==================
// Return the owning Security state and Exception level. Must only be called
// when SelfHostedTraceEnabled() is TRUE.

(SecurityState, bits(2)) TraceBufferOwner()
    assert HaveTraceBufferExtension() && SelfHostedTraceEnabled();

    SecurityState owning_ss;
    if HaveEL(EL3) then
        bits(3) state_bits;
        if HaveRME() then
            state_bits = MDCR_EL3.<NSTBE,NSTB>;
            if state_bits IN {'10x'} then
                // Reserved value
                (-, state_bits) = ConstrainUnpredictableBits(Unpredictable_RESERVEDNSxB, 3);
        else
            state_bits = '0' : MDCR_EL3.NSTB;

        case state_bits of
            when '00x' owning_ss = SS_Secure;
            when '01x' owning_ss = SS_NonSecure;
            when '11x' owning_ss = SS_Realm;
    else
        owning_ss = if SecureOnlyImplementation() then SS_Secure else SS_NonSecure;
    bits(2) owning_el;
    if HaveEL(EL2) && (owning_ss != SS_Secure || IsSecureEL2Enabled()) then
        owning_el = if MDCR_EL2.E2TB == '00' then EL2 else EL1;
    else
        owning_el = EL1;
    return (owning_ss, owning_el);
```

Library pseudocode for shared/trace/TraceBuffer/TraceBufferRunning

```
// TraceBufferRunning()
// ====================

boolean TraceBufferRunning()
    return TraceBufferEnabled() && TRBSR_EL1.S == '0';
```

Library pseudocode for shared/trace/selfhosted/EffectiveE0HTRE

```
// EffectiveE0HTRE()
// =================
// Returns effective E0HTRE value

bit EffectiveE0HTRE()
    return if ELUsingAArch32(EL2) then HTRFCR.E0HTRE else TRFCR_EL2.E0HTRE;
```

Library pseudocode for shared/trace/selfhosted/EffectiveE0TRE

```
// EffectiveE0TRE()
// ================
// Returns effective E0TRE value

bit EffectiveE0TRE()
    return if ELUsingAArch32(EL1) then TRFCR.E0TRE else TRFCR_EL1.E0TRE;
```

Library pseudocode for shared/trace/selfhosted/EffectiveE1TRE

```
// EffectiveE1TRE()
// ================
// Returns effective E1TRE value

bit EffectiveE1TRE()
    return if UsingAArch32() then TRFCR.E1TRE else TRFCR_EL1.E1TRE;
```

Library pseudocode for shared/trace/selfhosted/EffectiveE2TRE

```
// EffectiveE2TRE()
// ================
// Returns effective E2TRE value

bit EffectiveE2TRE()
    return if UsingAArch32() then HTRFCR.E2TRE else TRFCR_EL2.E2TRE;
```

Library pseudocode for shared/trace/selfhosted/SelfHostedTraceEnabled

```
// SelfHostedTraceEnabled()
// ========================
// Returns TRUE if Self-hosted Trace is enabled.

boolean SelfHostedTraceEnabled()
    if !(HaveTraceExt() && HaveSelfHostedTrace()) then return FALSE;
    if EDSCR.TFO == '0' then return TRUE;
    if HaveEL(EL3) then
        secure_trace_enable = if ELUsingAArch32(EL3) then SDCR.STE else MDCR_EL3.STE;
        if secure_trace_enable == '1' && !ExternalSecureNoninvasiveDebugEnabled() then
            return TRUE;
        if (HaveRME() && MDCR_EL3.RLTE == '1' &&
            !ExternalRealmNoninvasiveDebugEnabled()) then
            return TRUE;
    else
        if SecureOnlyImplementation() && !ExternalSecureNoninvasiveDebugEnabled() then
            return TRUE;

    return FALSE;
```

Library pseudocode for shared/trace/selfhosted/TraceAllowed

```
// TraceAllowed()
// ==============
// Returns TRUE if Self-hosted Trace is allowed in the given Exception level.

boolean TraceAllowed(bits(2) el)
    if !HaveTraceExt() then return FALSE;
    if SelfHostedTraceEnabled() then
        boolean trace_allowed;
        ss = SecurityStateAtEL(el);
        // Detect scenarios where tracing in this Security state is never allowed.
        case ss of
            when SS_NonSecure
                trace_allowed = TRUE;
            when SS_Secure
                bit trace_bit;
                if HaveEL(EL3) then
                    trace_bit = if ELUsingAArch32(EL3) then SDCR.STE else MDCR_EL3.STE;
                else
                    trace_bit = '1';
                trace_allowed = trace_bit == '1';
            when SS_Realm
                trace_allowed = MDCR_EL3.RLTE == '1';
            when SS_Root
                trace_allowed = FALSE;

        // Tracing is prohibited if the trace buffer owning security state is not the
        // current Security state or the owning Exception level is a lower Exception level.
        if HaveTraceBufferExtension() && TraceBufferEnabled() then
            (owning_ss, owning_el) = TraceBufferOwner();
            if (ss != owning_ss || UInt(owning_el) < UInt(el) ||
                (EffectiveTGE() == '1' && owning_el == EL1)) then
                trace_allowed = FALSE;

        bit TRE_bit;
        case el of
            when EL3  TRE_bit = if !HaveAArch64() then TRFCR.E1TRE else '0';
            when EL2  TRE_bit = EffectiveE2TRE();
            when EL1  TRE_bit = EffectiveE1TRE();
            when EL0
                if EffectiveTGE() == '1' then
                    TRE_bit = EffectiveE0HTRE();
                else
                    TRE_bit = EffectiveE0TRE();

        return trace_allowed && TRE_bit == '1';
    else
        return ExternalNoninvasiveDebugAllowed(el);
```

Library pseudocode for shared/trace/selfhosted/TraceContextIDR2

```
// TraceContextIDR2()
// ==================

boolean TraceContextIDR2()
    if !TraceAllowed(PSTATE.EL)|| !HaveEL(EL2) then return FALSE;
    return (!SelfHostedTraceEnabled() || TRFCR_EL2.CX == '1');
```

Library pseudocode for shared/trace/selfhosted/TraceSynchronizationBarrier

```
// Memory barrier instruction that preserves the relative order of memory accesses to System
// registers due to trace operations and other memory accesses to the same registers
TraceSynchronizationBarrier();
```

Library pseudocode for shared/trace/selfhosted/TraceTimeStamp

```
// TraceTimeStamp()
// ================

TimeStamp TraceTimeStamp()
    if SelfHostedTraceEnabled() then
        if HaveEL(EL2) then
            TS_el2 = TRFCR_EL2.TS;
            if !HaveECVExt() && TS_el2 == '10' then
                // Reserved value
                (-, TS_el2) = ConstrainUnpredictableBits(Unpredictable_EL2TIMESTAMP, 2);

            case TS_el2 of
                when '00'
                    // Falls out to check TRFCR_EL1.TS
                when '01'
                    return TimeStamp_Virtual;
                when '10'
                    assert HaveECVExt();  // Otherwise ConstrainUnpredictableBits removes this case
                    return TimeStamp_OffsetPhysical;
                when '11'
                    return TimeStamp_Physical;

        TS_el1 = TRFCR_EL1.TS;
        if TS_el1 == '00' || (!HaveECVExt() && TS_el1 == '10') then
            // Reserved value
            (-, TS_el1) = ConstrainUnpredictableBits(Unpredictable_EL1TIMESTAMP, 2);

        case TS_el1 of
            when '01'
                return TimeStamp_Virtual;
            when '10'
                assert HaveECVExt();
                return TimeStamp_OffsetPhysical;
            when '11'
                return TimeStamp_Physical;
            otherwise
                Unreachable();         // ConstrainUnpredictableBits removes this case
    else
        return TimeStamp_CoreSight;
```

Library pseudocode for shared/trace/system/IsTraceCorePowered

```
// Returns TRUE if the Trace Core Power Domain is powered up
boolean IsTraceCorePowered();
```

Library pseudocode for shared/translation/at/ATAccess

```
enumeration ATAccess {
    ATAccess_Read,
    ATAccess_Write,
    ATAccess_ReadPAN,
    ATAccess_WritePAN
};
```

Library pseudocode for shared/translation/at/EncodePARAttrs

```
// EncodePARAttrs()
// ================
// Convert orthogonal attributes and hints to 64-bit PAR ATTR field.

bits(8) EncodePARAttrs(MemoryAttributes memattrs)
    bits(8) result;

    if HaveMTEExt() && memattrs.tagged then
        result<7:0> = '11110000';
        return result;

    if memattrs.memtype == MemType_Device then
        result<7:4> = '0000';
        if memattrs.device == DeviceType_nGnRnE then
            result<3:0> = '0000';
        elsif memattrs.device == DeviceType_nGnRE then
            result<3:0> = '0100';
        elsif memattrs.device == DeviceType_nGRE then
            result<3:0> = '1000';
        else // DeviceType_GRE
            result<3:0> = '1100';
    else
        if memattrs.outer.attrs == MemAttr_WT then
            result<7:6> = if memattrs.outer.transient then '00' else '10';
            result<5:4> = memattrs.outer.hints;
        elsif memattrs.outer.attrs == MemAttr_WB then
            result<7:6> = if memattrs.outer.transient then '01' else '11';
            result<5:4> = memattrs.outer.hints;
        else // MemAttr_NC
            result<7:4> = '0100';

        if memattrs.inner.attrs == MemAttr_WT then
            result<3:2> = if memattrs.inner.transient then '00' else '10';
            result<1:0> = memattrs.inner.hints;
        elsif memattrs.inner.attrs == MemAttr_WB then
            result<3:2> = if memattrs.inner.transient then '01' else '11';
            result<1:0> = memattrs.inner.hints;
        else // MemAttr_NC
            result<3:0> = '0100';

    return result;
```

Library pseudocode for shared/translation/at/PAREncodeShareability

```
// PAREncodeShareability()
// =======================
// Derive 64-bit PAR SH field.

bits(2) PAREncodeShareability(MemoryAttributes memattrs)
    if (memattrs.memtype == MemType_Device ||
            (memattrs.inner.attrs == MemAttr_NC &&
             memattrs.outer.attrs == MemAttr_NC)) then
        // Force Outer-Shareable on Device and Normal Non-Cacheable memory
        return '10';

    case memattrs.shareability of
        when Shareability_NSH return '00';
        when Shareability_ISH return '11';
        when Shareability_OSH return '10';
```

Library pseudocode for shared/translation/at/TranslationStage

```
enumeration TranslationStage {
    TranslationStage_1,
    TranslationStage_12
};
```

Library pseudocode for shared/translation/attrs/DecodeDevice

```
// DecodeDevice()
// ==============
// Decode output Device type

DeviceType DecodeDevice(bits(2) device)
    case device of
        when '00'   return DeviceType_nGnRnE;
        when '01'   return DeviceType_nGnRE;
        when '10'   return DeviceType_nGRE;
        when '11'   return DeviceType_GRE;
```

Library pseudocode for shared/translation/attrs/DecodeLDFAttr

```
// DecodeLDFAttr()
// ===============
// Decode memory attributes using LDF (Long Descriptor Format) mapping

MemAttrHints DecodeLDFAttr(bits(4) attr)
    MemAttrHints ldfattr;

    if    attr IN {'x0xx'} then ldfattr.attrs = MemAttr_WT; // Write-through
    elsif attr == '0100' then ldfattr.attrs = MemAttr_NC;   // Non-cacheable
    elsif attr IN {'x1xx'} then ldfattr.attrs = MemAttr_WB; // Write-back
    else                        Unreachable();

    // Allocation hints are applicable only to cacheable memory.
    if ldfattr.attrs != MemAttr_NC then
        case attr<1:0> of
            when '00' ldfattr.hints = MemHint_No;  // No allocation hints
            when '01' ldfattr.hints = MemHint_WA;  // Write-allocate
            when '10' ldfattr.hints = MemHint_RA;  // Read-allocate
            when '11' ldfattr.hints = MemHint_RWA; // Read/Write allocate

    // The Transient hint applies only to cacheable memory with some allocation hints.
    if ldfattr.attrs != MemAttr_NC && ldfattr.hints != MemHint_No then
        ldfattr.transient = attr<3> == '0';

    return ldfattr;
```

Library pseudocode for shared/translation/attrs/DecodeSDFAttr

```
// DecodeSDFAttr()
// ===============
// Decode memory attributes using SDF (Short Descriptor Format) mapping

MemAttrHints DecodeSDFAttr(bits(2) rgn)
    MemAttrHints sdfattr;

    case rgn of
        when '00'                   // Non-cacheable (no allocate)
            sdfattr.attrs = MemAttr_NC;
        when '01'                   // Write-back, Read and Write allocate
            sdfattr.attrs = MemAttr_WB;
            sdfattr.hints = MemHint_RWA;
        when '10'                   // Write-through, Read allocate
            sdfattr.attrs = MemAttr_WT;
            sdfattr.hints = MemHint_RA;
        when '11'                   // Write-back, Read allocate
            sdfattr.attrs = MemAttr_WB;
            sdfattr.hints = MemHint_RA;

    sdfattr.transient = FALSE;

    return sdfattr;
```

Library pseudocode for shared/translation/attrs/DecodeShareability

```
// DecodeShareability()
// ===================
// Decode shareability of target memory region

Shareability DecodeShareability(bits(2) sh)
    case sh of
        when '10' return Shareability_OSH;
        when '11' return Shareability_ISH;
        when '00' return Shareability_NSH;
        otherwise
            case ConstrainUnpredictable(Unpredictable_Shareability) of
                when Constraint_OSH return Shareability_OSH;
                when Constraint_ISH return Shareability_ISH;
                when Constraint_NSH return Shareability_NSH;
```

Library pseudocode for shared/translation/attrs/EffectiveShareability

```
// EffectiveShareability()
// =======================
// Force Outer Shareability on Device and Normal iNCoNC memory

Shareability EffectiveShareability(MemoryAttributes memattrs)
    if (memattrs.memtype == MemType_Device ||
            (memattrs.inner.attrs == MemAttr_NC &&
             memattrs.outer.attrs == MemAttr_NC)) then
        return Shareability_OSH;
    else
        return memattrs.shareability;
```

Library pseudocode for shared/translation/attrs/IsS2ResultTagged

```
// IsS2ResultTagged()
// ==================
// Determine whether the combined output memory attributes of stage 1 and
// stage 2 indicate tagged memory

boolean IsS2ResultTagged(MemoryAttributes s2_memattrs, boolean s1_tagged)

    if !HaveMTE2Ext() then
        return FALSE;

    return (
        s1_tagged                                  &&
        (s2_memattrs.memtype == MemType_Normal)    &&
        (s2_memattrs.inner.attrs == MemAttr_WB)    &&
        (s2_memattrs.inner.hints == MemHint_RWA)   &&
        (!s2_memattrs.inner.transient)             &&
        (s2_memattrs.outer.attrs == MemAttr_WB)    &&
        (s2_memattrs.outer.hints == MemHint_RWA)   &&
        (!s2_memattrs.outer.transient)
    );
```

Library pseudocode for shared/translation/attrs/MAIRAttr

```
// MAIRAttr()
// ==========
// Retrieve the memory attribute encoding indexed in the given MAIR

bits(8) MAIRAttr(integer index, MAIRType mair)
    bit_index = 8 * index;
    return mair<bit_index+7:bit_index>;
```

Library pseudocode for shared/translation/attrs/NormalNCMemAttr

```
// NormalNCMemAttr()
// =================
// Normal Non-cacheable memory attributes

MemoryAttributes NormalNCMemAttr()
    MemAttrHints non_cacheable;
    non_cacheable.attrs = MemAttr_NC;

    MemoryAttributes nc_mematttrs;
    nc_mematttrs.memtype      = MemType_Normal;
    nc_mematttrs.outer        = non_cacheable;
    nc_mematttrs.inner        = non_cacheable;
    nc_mematttrs.shareability = Shareability_OSH;
    nc_mematttrs.tagged       = FALSE;

    return nc_mematttrs;
```

Library pseudocode for shared/translation/attrs/S1ConstrainUnpredictableRESMAIR

```
// S1ConstrainUnpredictableRESMAIR()
// =================================
// Determine whether a reserved value occupies MAIR_ELx.AttrN

boolean S1ConstrainUnpredictableRESMAIR(bits(8) attr, boolean s1aarch64)
    case attr of
        when '0000xx01' return !(s1aarch64 && HaveFeatXS());
        when '0000xxxx' return attr<1:0> != '00';
        when '01000000' return !(s1aarch64 && HaveFeatXS());
        when '10100000' return !(s1aarch64 && HaveFeatXS());
        when '11110000' return !(s1aarch64 && HaveMTE2Ext());
        when 'xxxx0000' return TRUE;
        otherwise       return FALSE;
```

Library pseudocode for shared/translation/attrs/S1DecodeMemAttrs

```
// S1DecodeMemAttrs()
// ==================
// Decode MAIR-format memory attributes assigned in stage 1

MemoryAttributes S1DecodeMemAttrs(bits(8) attr_in, bits(2) sh, boolean s1aarch64)
    bits(8) attr = attr_in;
    if S1ConstrainUnpredictableRESMAIR(attr, s1aarch64) then
        (-, attr) = ConstrainUnpredictableBits(Unpredictable_RESMAIR, 8);

    MemoryAttributes memattrs;
    case attr of
        when '0000xxxx' // Device memory
            memattrs.memtype = MemType_Device;
            memattrs.device  = DecodeDevice(attr<3:2>);
            memattrs.tagged  = FALSE;
            memattrs.xs      = if s1aarch64 then NOT attr<0> else '1';
        when '01000000'
            assert s1aarch64 && HaveFeatXS();
            memattrs.memtype = MemType_Normal;
            memattrs.tagged  = FALSE;
            memattrs.outer.attrs = MemAttr_NC;
            memattrs.inner.attrs = MemAttr_NC;
            memattrs.xs          = '0';

        when '10100000'
            assert s1aarch64 && HaveFeatXS();
            memattrs.memtype = MemType_Normal;
            memattrs.tagged  = FALSE;
            memattrs.outer.attrs     = MemAttr_WT;
            memattrs.outer.hints     = MemHint_RA;
            memattrs.outer.transient = FALSE;
            memattrs.inner.attrs     = MemAttr_WT;
            memattrs.inner.hints     = MemHint_RA;
            memattrs.inner.transient = FALSE;
            memattrs.xs              = '0';
        when '11110000' // Tagged memory
            assert s1aarch64 && HaveMTE2Ext();
            memattrs.memtype = MemType_Normal;
            memattrs.tagged  = TRUE;
            memattrs.outer.attrs     = MemAttr_WB;
            memattrs.outer.hints     = MemHint_RWA;
            memattrs.outer.transient = FALSE;
            memattrs.inner.attrs     = MemAttr_WB;
            memattrs.inner.hints     = MemHint_RWA;
            memattrs.inner.transient = FALSE;
            memattrs.xs              = '0';
        otherwise
            memattrs.memtype = MemType_Normal;
            memattrs.outer   = DecodeLDFAttr(attr<7:4>);
            memattrs.inner   = DecodeLDFAttr(attr<3:0>);
            memattrs.tagged  = FALSE;

            if (memattrs.inner.attrs == MemAttr_WB &&
                    memattrs.outer.attrs == MemAttr_WB) then
                memattrs.xs = '0';
            else
                memattrs.xs = '1';

    memattrs.shareability = DecodeShareability(sh);

    return memattrs;
```

Library pseudocode for shared/translation/attrs/S2CombineS1AttrHints

```
// S2CombineS1AttrHints()
// ======================
// Determine resultant Normal memory cacheability and allocation hints from
// combining stage 1 Normal memory attributes and stage 2 cacheability attributes.

MemAttrHints S2CombineS1AttrHints(MemAttrHints s1_attrhints, MemAttrHints s2_attrhints)
    MemAttrHints attrhints;

    if s1_attrhints.attrs == MemAttr_NC || s2_attrhints.attrs == MemAttr_NC then
        attrhints.attrs = MemAttr_NC;
    elsif s1_attrhints.attrs == MemAttr_WT || s2_attrhints.attrs == MemAttr_WT then
        attrhints.attrs = MemAttr_WT;
    else
        attrhints.attrs = MemAttr_WB;

    // Stage 2 does not assign any allocation hints
    // Instead, they are inherited from stage 1
    if attrhints.attrs != MemAttr_NC then
        attrhints.hints     = s1_attrhints.hints;
        attrhints.transient = s1_attrhints.transient;

    return attrhints;
```

Library pseudocode for shared/translation/attrs/S2CombineS1Device

```
// S2CombineS1Device()
// ===================
// Determine resultant Device type from combining output memory attributes
// in stage 1 and Device attributes in stage 2

DeviceType S2CombineS1Device(DeviceType s1_device, DeviceType s2_device)
    if s1_device == DeviceType_nGnRnE || s2_device == DeviceType_nGnRnE then
        return DeviceType_nGnRnE;
    elsif s1_device == DeviceType_nGnRE || s2_device == DeviceType_nGnRE then
        return DeviceType_nGnRE;
    elsif s1_device == DeviceType_nGRE || s2_device == DeviceType_nGRE then
        return DeviceType_nGRE;
    else
        return DeviceType_GRE;
```

Library pseudocode for shared/translation/attrs/S2CombineS1MemAttrs

```
// S2CombineS1MemAttrs()
// =====================
// Combine stage 2 with stage 1 memory attributes

MemoryAttributes S2CombineS1MemAttrs(MemoryAttributes s1_mematttrs,
                                     MemoryAttributes s2_mematttrs)
    MemoryAttributes mematttrs;

    if s1_mematttrs.memtype == MemType_Device && s2_mematttrs.memtype == MemType_Device then
        mematttrs.memtype = MemType_Device;
        mematttrs.device  = S2CombineS1Device(s1_mematttrs.device, s2_mematttrs.device);
    elsif s1_mematttrs.memtype == MemType_Device then    // S2 Normal, S1 Device
        mematttrs = s1_mematttrs;
    elsif s2_mematttrs.memtype == MemType_Device then    // S2 Device, S1 Normal
        mematttrs = s2_mematttrs;
    else                                                 // S2 Normal, S1 Normal
        mematttrs.memtype = MemType_Normal;
        mematttrs.inner   = S2CombineS1AttrHints(s1_mematttrs.inner, s2_mematttrs.inner);
        mematttrs.outer   = S2CombineS1AttrHints(s1_mematttrs.outer, s2_mematttrs.outer);

        mematttrs.tagged = IsS2ResultTagged(mematttrs, s1_mematttrs.tagged);

    mematttrs.shareability = S2CombineS1Shareability(s1_mematttrs.shareability,
                                                     s2_mematttrs.shareability);
    mematttrs.xs           = s2_mematttrs.xs;

    mematttrs.shareability = EffectiveShareability(mematttrs);
    return mematttrs;
```

Library pseudocode for shared/translation/attrs/S2CombineS1Shareability

```
// S2CombineS1Shareability()
// =========================
// Combine stage 2 shareability with stage 1

Shareability S2CombineS1Shareability(Shareability s1_shareability,
                                     Shareability s2_shareability)

    if (s1_shareability == Shareability_OSH ||
            s2_shareability == Shareability_OSH) then
        return Shareability_OSH;
    elsif (s1_shareability == Shareability_ISH ||
            s2_shareability == Shareability_ISH) then
        return Shareability_ISH;
    else
        return Shareability_NSH;
```

Library pseudocode for shared/translation/attrs/S2DecodeCacheability

```
// S2DecodeCacheability()
// ======================
// Determine the stage 2 cacheability for Normal memory

MemAttrHints S2DecodeCacheability(bits(2) attr)
    MemAttrHints s2attr;

    case attr of
        when '01' s2attr.attrs = MemAttr_NC;  // Non-cacheable
        when '10' s2attr.attrs = MemAttr_WT;  // Write-through
        when '11' s2attr.attrs = MemAttr_WB;  // Write-back
        otherwise // Constrained unpredictable
            case ConstrainUnpredictable(Unpredictable_S2RESMEMATTR) of
                when Constraint_NC s2attr.attrs = MemAttr_NC;
                when Constraint_WT s2attr.attrs = MemAttr_WT;
                when Constraint_WB s2attr.attrs = MemAttr_WB;

    // Stage 2 does not assign hints or the transient property
    // They are inherited from stage 1 if the result of the combination allows it
    s2attr.hints     = bits(2) UNKNOWN;
    s2attr.transient = boolean UNKNOWN;

    return s2attr;
```

Library pseudocode for shared/translation/attrs/S2DecodeMemAttrs

```
// S2DecodeMemAttrs()
// ==================
// Decode stage 2 memory attributes

MemoryAttributes S2DecodeMemAttrs(bits(4) attr, bits(2) sh, boolean s2aarch64)
    MemoryAttributes memattrs;

    case attr of
        when '00xx' // Device memory
            memattrs.memtype      = MemType_Device;
            memattrs.device       = DecodeDevice(attr<1:0>);
        otherwise   // Normal memory
            memattrs.memtype      = MemType_Normal;
            memattrs.outer        = S2DecodeCacheability(attr<3:2>);
            memattrs.inner        = S2DecodeCacheability(attr<1:0>);

    memattrs.shareability = DecodeShareability(sh);

    return memattrs;
```

Library pseudocode for shared/translation/attrs/WalkMemAttrs

```
// WalkMemAttrs()
// ==============
// Retrieve memory attributes of translation table walk

MemoryAttributes WalkMemAttrs(bits(2) sh, bits(2) irgn, bits(2) orgn)
    MemoryAttributes walkmemattrs;

    walkmemattrs.memtype      = MemType_Normal;
    walkmemattrs.shareability = DecodeShareability(sh);
    walkmemattrs.inner        = DecodeSDFAttr(irgn);
    walkmemattrs.outer        = DecodeSDFAttr(orgn);
    walkmemattrs.tagged       = FALSE;
    if (walkmemattrs.inner.attrs == MemAttr_WB &&
            walkmemattrs.outer.attrs == MemAttr_WB) then
        walkmemattrs.xs = '0';
    else
        walkmemattrs.xs = '1';

    return walkmemattrs;
```

Library pseudocode for shared/translation/faults/AlignmentFault

```
// AlignmentFault()
// ================

FaultRecord AlignmentFault(AccType acctype, boolean iswrite, boolean secondstage)
    FaultRecord fault;

    fault.statuscode  = Fault_Alignment;
    fault.acctype     = acctype;
    fault.write       = iswrite;
    fault.secondstage = secondstage;

    return fault;
```

Library pseudocode for shared/translation/faults/AsyncExternalAbort

```
// AsyncExternalAbort()
// ====================
// Return a fault record indicating an asynchronous External abort

FaultRecord AsyncExternalAbort(boolean parity, bits(2) errortype, bit extflag)
    FaultRecord fault;

    fault.statuscode  = if parity then Fault_AsyncParity else Fault_AsyncExternal;
    fault.extflag     = extflag;
    fault.errortype   = errortype;
    fault.acctype     = AccType_NORMAL;
    fault.secondstage = FALSE;
    fault.s2fs1walk   = FALSE;

    return fault;
```

Library pseudocode for shared/translation/faults/NoFault

```
// NoFault()
// =========
// Return a clear fault record indicating no faults have occured

FaultRecord NoFault()
    FaultRecord fault;

    fault.statuscode  = Fault_None;
    fault.acctype     = AccType_NORMAL;
    fault.secondstage = FALSE;
    fault.s2fs1walk   = FALSE;
    fault.gpcfs2walk  = FALSE;
    fault.gpcf        = GPCNoFault();

    return fault;
```

Library pseudocode for shared/translation/gpc/AbovePPS

```
// AbovePPS()
// ==========
// Returns TRUE if an address exceeds the range configured in GPCCR_EL3.PPS.

boolean AbovePPS(bits(52) address)
    pps = DecodePPS();
    if pps == 52 then
        return FALSE;

    return !IsZero(address<51:pps>);
```

Library pseudocode for shared/translation/gpc/DecodeGPTBlock

```
// DecodeGPTBlock()
// ================
// Validate and decode a GPT Block descriptor

(GPCF, GPTEntry) DecodeGPTBlock(PGSe pgs, bits(64) entry)
    assert entry<3:0> == GPT_Block;
    GPTEntry result;

    if !IsZero(entry<63:8>) then
        return (GPCF_Walk, GPTEntry UNKNOWN);

    if !GPIValid(entry<7:4>) then
        return (GPCF_Walk, GPTEntry UNKNOWN);

    result.gpi   = entry<7:4>;
    result.level = 0;

    // GPT information from a level 0 GPT Block descriptor is permitted
    // to be cached in a TLB as though the Block is a contiguous region
    // of granules each of the size configured in GPCCR_EL3.PGS.
    case pgs of
        when PGS_4KB  result.size = GPTRange_4KB;
        when PGS_16KB result.size = GPTRange_16KB;
        when PGS_64KB result.size = GPTRange_64KB;
        otherwise Unreachable();
    result.contig_size = GPTL0Size();

    return (GPCF_None, result);
```

Library pseudocode for shared/translation/gpc/DecodeGPTContiguous

```
// DecodeGPTContiguous()
// =====================
// Validate and decode a GPT Contiguous descriptor

(GPCF, GPTEntry) DecodeGPTContiguous(PGSe pgs, bits(64) entry)
    assert entry<3:0> == GPT_Contig;
    GPTEntry result;

    if !IsZero(entry<63:10>) then
        return (GPCF_Walk, result);

    result.gpi = entry<7:4>;
    if !GPIValid(result.gpi) then
        return (GPCF_Walk, result);

    case pgs of
        when PGS_4KB  result.size = GPTRange_4KB;
        when PGS_16KB result.size = GPTRange_16KB;
        when PGS_64KB result.size = GPTRange_64KB;
        otherwise Unreachable();

    case entry<9:8> of
        when '01' result.contig_size = GPTRange_2MB;
        when '10' result.contig_size = GPTRange_32MB;
        when '11' result.contig_size = GPTRange_512MB;
        otherwise return (GPCF_Walk, GPTEntry UNKNOWN);

    result.level = 1;

    return (GPCF_None, result);
```

Library pseudocode for shared/translation/gpc/DecodeGPTGranules

```
// DecodeGPTGranules()
// ===================
// Validate and decode a GPT Granules descriptor

(GPCF, GPTEntry) DecodeGPTGranules(PGSe pgs, integer index, bits(64) entry)
    GPTEntry result;

    for i = 0 to 15
        if !GPIValid(entry<i*4 +:4>) then
            return (GPCF_Walk, result);

    result.gpi = entry<index*4 +:4>;

    case pgs of
        when PGS_4KB  result.size = GPTRange_4KB;
        when PGS_16KB result.size = GPTRange_16KB;
        when PGS_64KB result.size = GPTRange_64KB;
        otherwise Unreachable();

    result.contig_size = result.size; // No contiguity
    result.level = 1;

    return (GPCF_None, result);
```

Library pseudocode for shared/translation/gpc/DecodeGPTTable

```
// DecodeGPTTable()
// ================
// Validate and decode a GPT Table descriptor

(GPCF, GPTTable) DecodeGPTTable(PGSe pgs, bits(64) entry)
    assert entry<3:0> == GPT_Table;
    GPTTable result;

    if !IsZero(entry<63:52,11:4>) then
        return (GPCF_Walk, GPTTable UNKNOWN);

    l0sz = GPTL0Size();
    integer p;
    case pgs of
        when PGS_4KB  p = 12;
        when PGS_16KB p = 14;
        when PGS_64KB p = 16;
        otherwise Unreachable();

    if !IsZero(entry<(l0sz-p)-2:12>) then
        return (GPCF_Walk, GPTTable UNKNOWN);

    case pgs of
        when PGS_4KB  result.address = entry<51:17>:Zeros(17);
        when PGS_16KB result.address = entry<51:15>:Zeros(15);
        when PGS_64KB result.address = entry<51:13>:Zeros(13);
        otherwise Unreachable();

    // The address must be within the range covered by the GPT
    if AbovePPS(result.address) then
        return (GPCF_AddressSize, GPTTable UNKNOWN);

    return (GPCF_None, result);
```

Library pseudocode for shared/translation/gpc/DecodePGS

```
// DecodePGS()
// ===========

PGSe DecodePGS(bits(2) pgs)
    case pgs of
        when '00' return PGS_4KB;
        when '10' return PGS_16KB;
        when '01' return PGS_64KB;
        otherwise Unreachable();
```

Library pseudocode for shared/translation/gpc/DecodePPS

```
// DecodePPS()
// ===========
// Size of region protected by the GPT, in bits.

integer DecodePPS()
    case GPCCR_EL3.PPS of
        when '000' return 32;
        when '001' return 36;
        when '010' return 40;
        when '011' return 42;
        when '100' return 44;
        when '101' return 48;
        when '110' return 52;
        otherwise Unreachable();
```

Library pseudocode for shared/translation/gpc/GPCFault

```
// GPCFault()
// ==========
// Constructs and returns a GPCF

GPCFRecord GPCFault(GPCF gpf, integer level)
    GPCFRecord fault;
    fault.gpf   = gpf;
    fault.level = level;
    return fault;
```

Library pseudocode for shared/translation/gpc/GPCNoFault

```
// GPCNoFault()
// ============
// Returns the default properties of a GPCF that does not represent a fault

GPCFRecord GPCNoFault()
    GPCFRecord result;
    result.gpf = GPCF_None;
    return result;
```

Library pseudocode for shared/translation/gpc/GPCRegistersConsistent

```
// GPCRegistersConsistent()
// ========================
// Returns whether the GPT registers are configured correctly.
// This returns false if any fields select a Reserved value.

boolean GPCRegistersConsistent()

    // Check for Reserved register values
    if GPCCR_EL3.PPS == '111' || DecodePPS() > AArch64.PAMax() then
        return FALSE;
    if GPCCR_EL3.PGS == '11' then
        return FALSE;
    if GPCCR_EL3.SH == '01' then
        return FALSE;

    // Inner and Outer Non-cacheable requires Outer Shareable
    if GPCCR_EL3.<ORGN, IRGN> == '0000' && GPCCR_EL3.SH != '10' then
        return FALSE;

    return TRUE;
```

Library pseudocode for shared/translation/gpc/GPICheck

```
// GPICheck()
// ==========
// Returns whether an access to a given physical address space is permitted
// given the configured GPI value.
// paspace: Physical address space of the access
// gpi: Value read from GPT for the access

boolean GPICheck(PASpace paspace, bits(4) gpi)

    case gpi of
        when GPT_NoAccess  return FALSE;
        when GPT_Secure    return paspace == PAS_Secure;
        when GPT_NonSecure return paspace == PAS_NonSecure;
        when GPT_Root      return paspace == PAS_Root;
        when GPT_Realm     return paspace == PAS_Realm;
        when GPT_Any       return TRUE;
        otherwise          Unreachable();
```

Library pseudocode for shared/translation/gpc/GPIIndex

```
// GPIIndex()
// ==========

integer GPIIndex(bits(52) pa)
    case DecodePGS(GPCCR_EL3.PGS) of
        when PGS_4KB  return UInt(pa<15:12>);
        when PGS_16KB return UInt(pa<17:14>);
        when PGS_64KB return UInt(pa<19:16>);
        otherwise Unreachable();
```

Library pseudocode for shared/translation/gpc/GPIValid

```
// GPIValid()
// ==========
// Returns whether a given value is a valid encoding for a GPI value

boolean GPIValid(bits(4) gpi)
    return gpi IN {GPT_NoAccess,
                   GPT_Secure,
                   GPT_NonSecure,
                   GPT_Root,
                   GPT_Realm,
                   GPT_Any};
```

Library pseudocode for shared/translation/gpc/GPTL0Size

```
// GPTL0Size()
// ===========
// Returns number of bits covered by a level 0 GPT entry

integer GPTL0Size()
    case GPCCR_EL3.L0GPTSZ of
        when '0000' return GPTRange_1GB;
        when '0100' return GPTRange_16GB;
        when '0110' return GPTRange_64GB;
        when '1001' return GPTRange_512GB;
        otherwise Unreachable();
    return 30;
```

Library pseudocode for shared/translation/gpc/GPTLevel0Index

```
// GPTLevel0Index()
// ================
// Compute the level 0 index based on input PA.

integer GPTLevel0Index(bits(52) pa)
    // Input address and index bounds
    pps = DecodePPS();
    l0sz = GPTL0Size();
    if pps <= l0sz then
        return 0;

    return UInt(pa<pps-1:l0sz>);
```

Library pseudocode for shared/translation/gpc/GPTLevel1Index

```
// GPTLevel1Index()
// ================
// Compute the level 1 index based on input PA.

integer GPTLevel1Index(bits(52) pa)
    // Input address and index bounds
    l0sz = GPTL0Size();
    case DecodePGS(GPCCR_EL3.PGS) of
        when PGS_4KB  return UInt(pa<l0sz-1:16>);
        when PGS_16KB return UInt(pa<l0sz-1:18>);
        when PGS_64KB return UInt(pa<l0sz-1:20>);
        otherwise Unreachable();
```

Library pseudocode for shared/translation/gpc/GPTWalk

```
// GPTWalk()
// =========
// Get the GPT entry for a given physical address, pa

(GPCFRecord, GPTEntry) GPTWalk(bits(52) pa, AccessDescriptor accdesc)
    // GPT base address
    bits(52) base;
    pgs = DecodePGS(GPCCR_EL3.PGS);

    // The level 0 GPT base address is aligned to the greater of:
    // * the size of the level 0 GPT, determined by GPCCR_EL3.{PPS, L0GPTSZ}.
    // * 4KB
    base = GPTBR_EL3.BADDR:Zeros(12);
    pps = DecodePPS();
    l0sz = GPTL0Size();
    integer alignment = Max((pps - l0sz) + 3, 12);
    base = base AND NOT ZeroExtend(Ones(alignment), 52);

    // Access attributes and address for GPT fetches
    AddressDescriptor gptaddrdesc;
    gptaddrdesc.fault    = NoFault();
    gptaddrdesc.memattrs = WalkMemAttrs(GPCCR_EL3.SH, GPCCR_EL3.ORGN, GPCCR_EL3.IRGN);

    // Address of level 0 GPT entry
    gptaddrdesc.paddress.paspace = PAS_Root;
    gptaddrdesc.paddress.address = base + GPTLevel0Index(pa) * 8;

    // Fetch L0GPT entry
    bits(64) level_0_entry;
    PhysMemRetStatus memstatus;
    (memstatus, level_0_entry) = PhysMemRead(gptaddrdesc, 8, accdesc);
    if IsFault(memstatus) then
        return (GPCFault(GPCF_EABT, 0), GPTEntry UNKNOWN);

    GPTEntry result;
    GPTTable table;
    GPCF gpf;
    case level_0_entry<3:0> of
        when GPT_Block
            // Decode the GPI value and return that
            (gpf, result) = DecodeGPTBlock(pgs, level_0_entry);
            result.pa = pa;
            return (GPCFault(gpf, 0), result);
        when GPT_Table
            // Decode the table entry and continue walking
            (gpf, table) = DecodeGPTTable(pgs, level_0_entry);
            if gpf != GPCF_None then
                return (GPCFault(gpf, 0), GPTEntry UNKNOWN);
        otherwise
            // GPF - invalid encoding
            return (GPCFault(GPCF_Walk, 0), GPTEntry UNKNOWN);

    // Must be a GPT Table entry
    assert level_0_entry<3:0> == GPT_Table;

    // Address of level 1 GPT entry
    offset = GPTLevel1Index(pa) * 8;
    gptaddrdesc.paddress.address = table.address + offset;

    // Fetch L1GPT entry
    bits(64) level_1_entry;
    (memstatus, level_1_entry) = PhysMemRead(gptaddrdesc, 8, accdesc);
    if IsFault(memstatus) then
        return (GPCFault(GPCF_EABT, 1), GPTEntry UNKNOWN);

    case level_1_entry<3:0> of
        when GPT_Contig
            (gpf, result) = DecodeGPTContiguous(pgs, level_1_entry);
        otherwise
            gpi_index = GPIIndex(pa);
```

```
            (gpf, result) = DecodeGPTGranules(pgs, gpi_index, level_1_entry);

    if gpf != GPCF_None then
        return (GPCFault(gpf, 1), GPTEntry UNKNOWN);

    result.pa = pa;
    return (GPCNoFault(), result);
```

Library pseudocode for shared/translation/gpc/GranuleProtectionCheck

```
// GranuleProtectionCheck()
// ========================
// Returns whether a given access is permitted, according to the
// granule protection check.
// addrdesc and accdesc describe the access to be checked.

GPCFRecord GranuleProtectionCheck(AddressDescriptor addrdesc, AccessDescriptor accdesc)

    assert HaveRME();
    // The address to be checked
    address = addrdesc.paddress;

    // Bypass mode - all accesses pass
    if GPCCR_EL3.GPC == '0' then
        return GPCNoFault();

    // Configuration consistency check
    if !GPCRegistersConsistent() then
        return GPCFault(GPCF_Walk, 0);

    // Input address size check
    if AbovePPS(address.address) then
        if address.paspace == PAS_NonSecure then
            return GPCNoFault();
        else
            return GPCFault(GPCF_Fail, 0);

    // GPT base address size check
    bits(52) gpt_base = GPTBR_EL3.BADDR:Zeros(12);
    if AbovePPS(gpt_base) then
        return GPCFault(GPCF_AddressSize, 0);

    // GPT lookup
    (gpcf, gpt_entry) = GPTWalk(address.address, accdesc);
    if gpcf.gpf != GPCF_None then
        return gpcf;

    // Check input physical address space against GPI
    permitted = GPICheck(address.paspace, gpt_entry.gpi);

    if !permitted then
        gpcf = GPCFault(GPCF_Fail, gpt_entry.level);
        return gpcf;

    // Check passed

    return GPCNoFault();
```

Library pseudocode for shared/translation/gpc/PGS

```
enumeration PGSe {
    PGS_4KB,
    PGS_16KB,
    PGS_64KB
};
```

Library pseudocode for shared/translation/gpc/Table

```
constant bits(4) GPT_NoAccess  = '0000';
constant bits(4) GPT_Table     = '0011';
constant bits(4) GPT_Block     = '0001';
constant bits(4) GPT_Contig    = '0001';
constant bits(4) GPT_Secure    = '1000';
constant bits(4) GPT_NonSecure = '1001';
constant bits(4) GPT_Root      = '1010';
constant bits(4) GPT_Realm     = '1011';
constant bits(4) GPT_Any       = '1111';
constant integer GPTRange_4KB   = 12;
constant integer GPTRange_16KB  = 14;
constant integer GPTRange_64KB  = 16;
constant integer GPTRange_2MB   = 21;
constant integer GPTRange_32MB  = 25;
constant integer GPTRange_512MB = 29;
constant integer GPTRange_1GB   = 30;
constant integer GPTRange_16GB  = 34;
constant integer GPTRange_64GB  = 36;
constant integer GPTRange_512GB = 39;

type GPTTable is (
    bits(52) address        // Base address of next table
)

type GPTEntry is (
    bits(4)  gpi,           // GPI value for this region
    integer  size,          // Region size
    integer  contig_size,   // Contiguous region size
    integer  level,         // Level of GPT lookup
    bits(52) pa             // PA uniquely identifying the GPT entry
)
```

Library pseudocode for shared/translation/translation/S1TranslationRegime

```
// S1TranslationRegime()
// =====================
// Stage 1 translation regime for the given Exception level

bits(2) S1TranslationRegime(bits(2) el)
    if el != EL0 then
        return el;
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) && SCR.NS == '0' then
        return EL3;
    elsif HaveVirtHostExt() && ELIsInHost(el) then
        return EL2;
    else
        return EL1;

// S1TranslationRegime()
// =====================
// Returns the Exception level controlling the current Stage 1 translation regime. For the most
// part this is unused in code because the System register accessors (SCTLR[], etc.) implicitly
// return the correct value.

bits(2) S1TranslationRegime()
    return S1TranslationRegime(PSTATE.EL);
```

Library pseudocode for shared/translation/vmsa/AddressDescriptor

```
type AddressDescriptor is (
    FaultRecord      fault,      // fault.statuscode indicates whether the address is valid
    MemoryAttributes mematttrs,
    FullAddress      paddress,
    bits(64)         vaddress
)

constant integer FINAL_LEVEL = 3;
```

Library pseudocode for shared/translation/vmsa/ContiguousSize

```
// ContiguousSize()
// ================
// Return the number of entries log 2 marking a contiguous output range

integer ContiguousSize(TGx tgx, integer level)
    case tgx of
        when TGx_4KB
            assert level IN {1, 2, 3};
            return 4;
        when TGx_16KB
            assert level IN {2, 3};
            return if level == 2 then 5 else 7;
        when TGx_64KB
            assert level IN {2, 3};
            return 5;
```

Library pseudocode for shared/translation/vmsa/CreateAddressDescriptor

```
// CreateAddressDescriptor()
// =========================
// Set internal members for address descriptor type to valid values

AddressDescriptor CreateAddressDescriptor(bits(64) va, FullAddress pa,
                                          MemoryAttributes mematttrs)
    AddressDescriptor addrdesc;

    addrdesc.paddress = pa;
    addrdesc.vaddress = va;
    addrdesc.mematttrs = mematttrs;
    addrdesc.fault    = NoFault();

    return addrdesc;
```

Library pseudocode for shared/translation/vmsa/CreateFaultyAddressDescriptor

```
// CreateFaultyAddressDescriptor()
// ===============================
// Set internal members for address descriptor type with values indicating error

AddressDescriptor CreateFaultyAddressDescriptor(bits(64) va, FaultRecord fault)
    AddressDescriptor addrdesc;

    addrdesc.vaddress = va;
    addrdesc.fault    = fault;

    return addrdesc;
```

Library pseudocode for shared/translation/vmsa/DescriptorType

```
enumeration DescriptorType {
    DescriptorType_Table,
    DescriptorType_Block,
    DescriptorType_Page,
    DescriptorType_Invalid
};
```

Library pseudocode for shared/translation/vmsa/Domains

```
constant bits(2) Domain_NoAccess = '00';
constant bits(2) Domain_Client   = '01';
constant bits(2) Domain_Manager  = '11';
```

Library pseudocode for shared/translation/vmsa/FetchDescriptor

```
// FetchDescriptor()
// =================
// Fetch a translation table descriptor

(FaultRecord, bits(N)) FetchDescriptor(bit ee, AddressDescriptor walkaddress, FaultRecord fault_in, integ
    // 32-bit descriptors for AArch32 Short-descriptor format
    // 64-bit descriptors for AArch64 or AArch32 Long-descriptor format
    assert N == 32 || N == 64;
    bits(N) descriptor;
    FaultRecord fault = fault_in;
    AccessDescriptor walkacc;

    walkacc.acctype = AccType_TTW;
    // MPAM PARTID for translation table walk is determined by the access invoking the translation
    walkacc.mpam    = GenMPAMcurEL(fault.acctype);

    if HaveRME() then
        fault.gpcf = GranuleProtectionCheck(walkaddress, walkacc);
        if fault.gpcf.gpf != GPCF_None then
            fault.statuscode = Fault_GPCFOnWalk;
            fault.paddress   = walkaddress.paddress;
            fault.gpcfs2walk = fault.secondstage;
            return (fault, bits(N) UNKNOWN);

    PhysMemRetStatus memstatus;
    (memstatus, descriptor) = PhysMemRead(walkaddress, N DIV 8, walkacc);
    if IsFault(memstatus) then
        fault = HandleExternalTTWAbort(memstatus, fault.write, walkaddress,
                                       walkacc, N DIV 8, fault);
        if IsFault(fault.statuscode) then
            return (fault, bits(N) UNKNOWN);

    if ee == '1' then
        descriptor = BigEndianReverse(descriptor);

    return (fault, descriptor);
```

Library pseudocode for shared/translation/vmsa/HasUnprivileged

```
// HasUnprivileged()
// =================
// Returns whether a translation regime serves EL0 as well as a higher EL

boolean HasUnprivileged(Regime regime)
    return (regime IN {
        Regime_EL20,
        Regime_EL30,
        Regime_EL10
    });
```

Library pseudocode for shared/translation/vmsa/IsAtomicRW

```
// IsAtomicRW()
// ============
// Is the access an atomic operation?

boolean IsAtomicRW(AccType acctype)
    return acctype IN {
                AccType_ATOMICRW,
                AccType_ORDEREDRW,
                AccType_ORDEREDATOMICRW
           };
```

Library pseudocode for shared/translation/vmsa/Regime

```
enumeration Regime {
    Regime_EL3,             // EL3
    Regime_EL30,            // EL3&0 (PL1&0 when EL3 is AArch32)
    Regime_EL2,             // EL2
    Regime_EL20,            // EL2&0
    Regime_EL10             // EL1&0
};
```

Library pseudocode for shared/translation/vmsa/RegimeUsingAArch32

```
// RegimeUsingAArch32()
// ====================
// Determine if the EL controlling the regime executes in AArch32 state

boolean RegimeUsingAArch32(Regime regime)
    case regime of
        when Regime_EL10 return ELUsingAArch32(EL1);
        when Regime_EL30 return TRUE;
        when Regime_EL20 return FALSE;
        when Regime_EL2  return ELUsingAArch32(EL2);
        when Regime_EL3  return FALSE;
```

Library pseudocode for shared/translation/vmsa/S1TTWParams

```
type S1TTWParams is (
// A64-VMSA exclusive parameters
    bit         ha,         // TCR_ELx.HA
    bit         hd,         // TCR_ELx.HD
    bit         tbi,        // TCR_ELx.TBI{x}
    bit         tbid,       // TCR_ELx.TBID{x}
    bit         nfd,        // TCR_EL1.NFDx or TCR_EL2.NFDx when HCR_EL2.E2H == '1'
    bit         e0pd,       // TCR_EL1.E0PDx or TCR_EL2.E0PDx when HCR_EL2.E2H == '1'
    bit         ds,         // TCR_ELx.DS
    bits(3)     ps,         // TCR_ELx.{I}PS
    bits(6)     txsz,       // TCR_ELx.TxSZ
    bit         epan,       // SCTLR_EL1.EPAN or SCTLR_EL2.EPAN when HCR_EL2.E2H == '1'
    bit         dct,        // HCR_EL2.DCT
    bit         nv1,        // HCR_EL2.NV1
    bit         cmow,       // SCTLR_EL1.CMOW or SCTLR_EL2.CMOW when HCR_EL2.E2H == '1'

// A32-VMSA exclusive parameters
    bits(3)     t0sz,       // TTBCR.T0SZ
    bits(3)     t1sz,       // TTBCR.T1SZ
    bit         uwxn,       // SCTLR.UWXN

// Parameters common to both A64-VMSA & A32-VMSA (A64/A32)
    TGx         tgx,        // TCR_ELx.TGx      / Always TGx_4KB
    bits(2)     irgn,       // TCR_ELx.IRGNx    / TTBCR.IRGNx or HTCR.IRGN0
    bits(2)     orgn,       // TCR_ELx.ORGNx    / TTBCR.ORGNx or HTCR.ORGN0
    bits(2)     sh,         // TCR_ELx.SHx      / TTBCR.SHx or HTCR.SH0
    bit         hpd,        // TCR_ELx.HPD{x}   / TTBCR2.HPDx or HTCR.HPD
    bit         ee,         // SCTLR_ELx.EE     / SCTLR.EE or HSCTLR.EE
    bit         wxn,        // SCTLR_ELx.WXN    / SCTLR.WXN or HSCTLR.WXN
    bit         ntlsmd,     // SCTLR_ELx.nTLSMD / SCTLR.nTLSMD or HSCTLR.nTLSMD
    bit         dc,         // HCR_EL2.DC       / HCR.DC
    bit         sif,        // SCR_EL3.SIF      / SCR.SIF
    MAIRType    mair        // MAIR_ELx         / MAIR1:MAIR0 or HMAIR1:HMAIR0
)
```

Library pseudocode for shared/translation/vmsa/S2TTWParams

```
type S2TTWParams is (
// A64-VMSA exclusive parameters
    bit         ha,     // VTCR_EL2.HA
    bit         hd,     // VTCR_EL2.HD
    bit         sl2,    // V{S}TCR_EL2.SL2
    bit         ds,     // VTCR_EL2.DS
    bit         sw,     // VSTCR_EL2.SW
    bit         nsw,    // VTCR_EL2.NSW
    bit         sa,     // VSTCR_EL2.SA
    bit         nsa,    // VTCR_EL2.NSA
    bits(3)     ps,     // VTCR_EL2.PS
    bits(6)     txsz,   // V{S}TCR_EL2.T0SZ
    bit         fwb,    // HCR_EL2.PTW
    bit         cmow,   // HCRX_EL2.CMOW

// A32-VMSA exclusive parameters
    bit         s,      // VTCR.S
    bits(4)     t0sz,   // VTCR.T0SZ

// Parameters common to both A64-VMSA & A32-VMSA if implemented (A64/A32)
    TGx         tgx,    // V{S}TCR_EL2.TG0  / Always TGx_4KB
    bits(2)     sl0,    // V{S}TCR_EL2.SL0  / VTCR.SL0
    bits(2)     irgn,   // VTCR_EL2.IRGN0   / VTCR.IRGN0
    bits(2)     orgn,   // VTCR_EL2.ORGN0   / VTCR.ORGN0
    bits(2)     sh,     // VTCR_EL2.SH0     / VTCR.SH0
    bit         ee,     // SCTLR_EL2.EE     / HSCTLR.EE
    bit         ptw,    // HCR_EL2.PTW      / HCR.PTW
    bit         vm      // HCR_EL2.VM       / HCR.VM
)
```

Library pseudocode for shared/translation/vmsa/SDFType

```
enumeration SDFType {
    SDFType_Table,
    SDFType_Invalid,
    SDFType_Supersection,
    SDFType_Section,
    SDFType_LargePage,
    SDFType_SmallPage
};
```

Library pseudocode for shared/translation/vmsa/SecurityStateForRegime

```
// SecurityStateForRegime()
// ========================
// Return the Security State of the given translation regime

SecurityState SecurityStateForRegime(Regime regime)
    case regime of
        when Regime_EL3     return SecurityStateAtEL(EL3);
        when Regime_EL30    return SS_Secure; // A32 EL3 is always Secure
        when Regime_EL2     return SecurityStateAtEL(EL2);
        when Regime_EL20    return SecurityStateAtEL(EL2);
        when Regime_EL10    return SecurityStateAtEL(EL1);
```

Library pseudocode for shared/translation/vmsa/StageOA

```
// StageOA()
// =========
// Given the final walk state (a page or block descriptor), map the untranslated
// input address bits to the output address

FullAddress StageOA(bits(64) ia, TGx tgx, TTWState walkstate)
    // Output Address
    FullAddress oa;
    integer csize;

    tsize = TranslationSize(tgx, walkstate.level);
    if walkstate.contiguous == '1' then
        csize = ContiguousSize(tgx, walkstate.level);
    else
        csize = 0;

    ia_msb = tsize + csize;
    oa.paspace = walkstate.baseaddress.paspace;
    oa.address = walkstate.baseaddress.address<51:ia_msb>:ia<ia_msb-1:0>;

    return oa;
```

Library pseudocode for shared/translation/vmsa/TGx

```
enumeration TGx {
    TGx_4KB,
    TGx_16KB,
    TGx_64KB
};
```

Library pseudocode for shared/translation/vmsa/TGxGranuleBits

```
// TGxGranuleBits()
// ================
// Retrieve the address size, in bits, of a granule

integer TGxGranuleBits(TGx tgx)
    case tgx of
        when TGx_4KB  return 12;
        when TGx_16KB return 14;
        when TGx_64KB return 16;
```

Library pseudocode for shared/translation/vmsa/TLBContext

```
type TLBContext is (
    SecurityState ss,
    Regime        regime,
    bits(16)      vmid,
    bits(16)      asid,
    bit           nG,
    PASpace       ipaspace, // Used in stage 2 lookups & invalidations only
    boolean       includes_s1,
    boolean       includes_s2,
    boolean       includes_gpt,
    bits(64)      ia,       // Input Address
    TGx           tg,
    bit           cnp,
    bit           xs        // XS attribute (FEAT_XS)
)
```

Library pseudocode for shared/translation/vmsa/TLBRecord

```
type TLBRecord is (
    TLBContext  context,
    TTWState    walkstate,
    integer     blocksize,    // Number of bits directly mapped from IA to OA
    integer     contigsize,   // Number of entries log 2 marking a contiguous output range
    bits(64)    s1descriptor, // Stage 1 leaf descriptor in memory (valid if the TLB caches stage 1)
    bits(64)    s2descriptor  // Stage 2 leaf descriptor in memory (valid if the TLB caches stage 2)
)
```

Library pseudocode for shared/translation/vmsa/TTWState

```
type TTWState is (
    boolean             istable,
    integer             level,
    FullAddress         baseaddress,
    bit                 contiguous,
    bit                 nG,
    bit                 guardedpage,
    SDFType             sdftype, // AArch32 Short-descriptor format walk only
    bits(4)             domain,  // AArch32 Short-descriptor format walk only
    MemoryAttributes    memattrs,
    Permissions         permissions
)
```

Library pseudocode for shared/translation/vmsa/TranslationRegime

```
// TranslationRegime()
// ===================
// Select the translation regime given the target EL and PE state

Regime TranslationRegime(bits(2) el_in, AccType acctype)

    bits(2) el;
    case acctype of
        when AccType_NV2REGISTER
            assert EL2Enabled() && el_in == EL1;
            el = EL2;
        otherwise
            el = el_in;

    if el == EL3 then
        return if ELUsingAArch32(EL3) then Regime_EL30 else Regime_EL3;
    elsif el == EL2 then
        return if ELIsInHost(EL2) then Regime_EL20 else Regime_EL2;
    elsif el == EL1 then
        return Regime_EL10;
    elsif el == EL0 then
        if CurrentSecurityState() == SS_Secure && ELUsingAArch32(EL3) then
            return Regime_EL30;
        elsif ELIsInHost(EL0) then
            return Regime_EL20;
        else
            return Regime_EL10;
    else
        Unreachable();
```

Library pseudocode for shared/translation/vmsa/TranslationSize

```
// TranslationSize()
// =================
// Compute the number of bits directly mapped from the input address
// to the output address

integer TranslationSize(TGx tgx, integer level)
    granulebits = TGxGranuleBits(tgx);
    blockbits   = (FINAL_LEVEL - level) * (granulebits - 3);

    return granulebits + blockbits;
```

Library pseudocode for shared/translation/vmsa/UseASID

```
// UseASID()
// =========
// Determine whether the translation context for the access requires ASID or is a global entry

boolean UseASID(TLBContext access)
    return HasUnprivileged(access.regime);
```

Library pseudocode for shared/translation/vmsa/UseVMID

```
// UseVMID()
// =========
// Determine whether the translation context for the access requires VMID to match a TLB entry

boolean UseVMID(TLBContext access)
    return access.regime == Regime_EL10 && EL2Enabled();
```

Library pseudocode for shared/translation/vmsa/VARange

```
enumeration VARange {
    VARange_LOWER,
    VARange_UPPER
};
```

Internal version only: isa v33.23junrel, AdvSIMD v29.1, pseudocode v2022-06_rel, sve v2022-06_rel ; Build timestamp: 2022-07-06T09:51